Efficient and Accurate homomorphic comparisons

Martin Zuber and Olive Chakraborty

CEA-LIST, 91190 Gif-sur-Yvette

martin.zuber@cea.fr, olive.chakraborty@cea.fr

May 20, 2022

Abstract

We design and implement a new efficient and accurate Fully homomorphic argmin/min or argmax/max comparison operator, which finds its application in numerous real-world use cases as a classifier. In particular we propose two versions of our algorithms using different tools from TFHE's functional bootstrapping toolkit. Our algorithm scales to any number of input data points with linear time complexity and logarithmic noise-propagation. Our algorithm is the fastest on the market for non-parallel comparisons with a high degree of accuracy and precision. For MNIST and SVHN datasets, which work under the PATE framework, using our algorithm, we achieve an accuracy of around 99.95% for both.

Keywords: Homomorphic Encryption, Argmin, TFHE Encryption Scheme, Bootstrapping, PATE framework, Nearest-Neighbour.

1 Introduction

Comparing two or more values is a very simple task in the clear domain and is a part of an incalculable amount of algorithms from "traditional" statistical analysis tools to the most up-to-date machine learning algorithms. For this reason, achieving an efficient and precise private comparison has been at the forefront of privacy-preserving computation research for decades.

Indeed, that was precisely the matter addressed in Yao's original millionaires' problem [19] which discussed how two millionaires could determine who is richer while keeping their actual wealth private. From that work spawned the rich and ever-evolving research around Multi-Party Computation (MPC). MPC requires at least a few rounds of interactions between parties that want to compute a result privately. This is in contrast with Homomorphic Encryption (HE) which requires no interaction.

HE has had its limitations since its theoretical inception in 1978 [16]. The idea is for an untrusted party to be able to compute over encrypted data with no

need for either the secret-key or interactions with the secret-key holder. HE schemes up to the end of the 2000s were either limited in terms of multiplicative depth or could only add and subtract ciphertexts [13]. Gentry's 2009 article and his Ph.D. thesis [9, 10] laid the groundwork for HE to increase its scope to more complex applications such as the one that interest us here. Since then, a host of different HE cryptosystems and applications have been developed by the cryptographic community. Among them, one of the most important applications is the computation of the minimum or the maximum from a collection of encrypted data. This application can be easily seen in numerous use cases : identify the closest model to an external model from the database; identify the model which receives the highest number of votes,.. etc.

1.1 Scope

We place ourselves in the simple context of an entity ("the client") holding a number of indexed values $x_1, \ldots, x_n$. Using a homomorphic encryption scheme, it encrypts these values $([x_1], \ldots, [x_n])$ and sends them to "the server", an untrusted entity. The server computes an argmin and min operation (alternatively argmax/max), returning an encryption of both the minimum (alternatively maximum) value among the x_i and its index. The server should do this without interacting with the client at any point, save for receiving possible evaluation keys.

While this scope may not seem very applicable in itself, solving this problem opens the door to a number of real-world use-cases. Some of those are: the case of a 1-Nearest Neighbour computation as in [21]; the case of an embedding-based neural network evaluation as in [20]; the case of a distributed learning algorithm as in the case of the PATE framework [14], to name a few. The particular case of PATE framework consists of a "student" sending unlabelled data to several "teachers", all with their own machine learning model that can classify that data. The teachers then vote for the label that should apply through the use of a third party called the "aggregator". This aggregator then sends the results of the vote to the student who learned labels for an unlabelled dataset without ever accessing the teacher's models. This framework can be improved as in [17] by adding both Differential Privacy and HE. The teachers can encrypt their votes using the student's HE public key and the aggregator can compute an argmax over the encrypted teacher votes. The result is sent to the student for decryption.

1.2 Prior work

Some of the work that has been done on the subject only addresses the original millionaires' problem by comparing two integer and returning the index of the biggest one. [18] is such an article. It claims to have the best times for a 1 to 1 bit-wise comparison of two integer on the cloud. Since they don't allow the for selection of the maximum value, their method can't trivially be expanded to

any number of values to compare. This reduces the range of applications and does not fit the scope that we defined for ourselves.

Modern work that fits in our scope can be best represented by two recent articles. [21] uses the TFHE cryptosystem and its potent bootstrapping operation. They build a fully homomorphic k-Nearest Neighbor operator (a generalization of the argmin problem with the addition of a distance computation) that has a quadratic complexity and therefore scales poorly in terms of time performance. This is because they use a "league" method where every input is compared to every other one first and then results are compiled. [12] was published simultaneously and combines a "tournament" method (where every input is compared to another input and the winner goes on to the next round) with the league method to solve the same problem. A third method to which we compare our results is the use of the original TFHE MUX bootstrapping gates from [6]. They can be used very easily to build any bit-wise computation, and the min/argmin problem is not an exception.

Among the distinctions one can make between these three methods, several are worth highlighting: the option for a batched computation, which reduces performance overhead significantly in some use-cases; whether the computation is fully homomorphic (the parameters do not depend on the number of inputs) or levelled is important when considering the scalability of a method; whether only the min, the argmin, or both can be obtained. Table 1 presents these characteristics for every method cited and our own proposed alternative.

	our work	[21]	[12]	[6]
L/F	L	F	L	F
batching	✗	✗	✓	✗
min/argmin	m/a	a	m/a	m

Table 1: A table comparing previous work to our own. The different lines show whether the algorithms are using levelled (L) or full HE (F), whether they can accommodate for batching and whether the algorithm outputs just the argmin (a) and/or the min (m).

1.3 Our Contributions

Table 2 compares our performance results for an min and argmin computation over N 8-bit integers. We compare ourselves both to the best existing work [12] and to the TFHE bitwise implementation. In the table, it is clear that our algorithm is much faster than TFHE's simple bitwise approach. At first glance, our algorithm is also much faster than that of [12]. This is not true for all cases. Indeed, by using BGV's batching, they can compute a min over a number of different integer arrays simultaneously. For this computation in particular they have 5220 ciphertext slots whether they use them or not. One therefore computes 5220 one-to-one comparisons as quickly as a single one-to-one comparison. While [12] and TFHE's bitwise approach allow for an exact comparison, our

work only provides an exact comparison over 4 bit integers (though still with 8 bits of precision for the final min value). This difference between accuracy and precision is explained in fine detail in Section 3.4 and the impact that our lower precision incurs is expanded upon in Section 4, showing that it can be used very effectively in real-world and state-of-the-art settings. Whether our algorithm is better therefore depends on the number of simultaneous comparisons one needs to make and the accuracy needed.

N	our work (s)	[12] (s)	[12] (amortized in ms)	[6] (s)
2	0.17	12.3	2.37	1.5
4	0.52	50.6	9.68	4.3
8	1.2	151	29.1	10.2
16	2.6	387	74.1	21.8
32	5.3	884	169	45.1
64	10.8	2112	405	91.7

Table 2: A table comparing previous work to our own on an exact min and/or argmin computation over N 8-bit integers. The amortized time by [12] is obtained because they run the comparison in parallel over arrays of size 5220 (see their Table 4). The bit-wise time is extrapolated assuming a 26ms MUX gate time from [6].

We argue that, although [12] proposes an algorithm that can be very efficient in some cases, it lacks malleability. A comparison between 64 integers takes 35 minutes whether we do 5220 of them or a single one. Our algorithm is less efficient for very large numbers of simultaneous computations but much more efficient for a single min/argmin computation whatever the number of values N. Furthermore, our method allows for a much bigger pool of inputs, while [12] restrict themselves to a maximum of 64 values to compare.

1.4 Paper outline

The organization of the article is as follows : Section 2 lays the underlying notations, definitions and details of the TFHE encryption scheme. We describe the various types of ciphertexts which are used along with the details of the various procedures like bootstrapping and key-switching. This is followed by the description of our algorithms in Section 3. We also provide a thorough noise analysis of algorithms and give a detailed analysis of the algorithm from its accuracy and precision point of view. Section 4 details the performance of our algorithms which includes its application to the PATE use-case.

2 Technical Background

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 (normal) distribution of mean μ and variance σ^2. The Gaussian distribution is the probability distribution with density:

$$f_{\mu,\sigma^2}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

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 sub-ring of polynomials with binary coefficients is $\mathbb{B}_N[X] = \mathbb{B}[X]/(X^N + 1)$ ($\mathbb{B} = \{0, 1\}$).

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.

Negacyclic functions are anti-periodic functions with period p: verifying $f(x) = -f(x + p)$. For example `sine` is anti-periodic with period π and periodic with period 2π.

2.2 TFHE encryption scheme

The TFHE encryption scheme was proposed in 2016 [3]. It improves the FHEW cryptosystem [8] and introduces the TLWE problem as an adaptation of the LWE problem to $\mathbb{T}$. It was updated later in [4] and both works were then unified in [6]. The TFHE scheme is implemented as the TFHE library [5]. We refer to the original articles for details on the TFHE cryptosystem. Here, we only give a high-level overview, one that is as high as possible for comprehension by the reader, but in-depth enough so that our results and their nuances can be understood. 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 encryption of $m \in \mathcal{M} \subset \mathbb{T}$ if $\boldsymbol{a} \xleftarrow{\$} \mathbb{T}^n$ and $b \in \mathbb{T}$ verifies $b = \langle \boldsymbol{a}, \boldsymbol{s} \rangle + m + e$, where $\boldsymbol{s} \xleftarrow{\$} \mathbb{B}^n$ is the secret key, and $e \xleftarrow{\mathcal{N}(0,\sigma^2)} \mathbb{T}$ is the noise introduced in the ciphertext.

- **TRLWE Sample:** a pair $(\boldsymbol{a}, b) \in \mathbb{T}_N[X]^k \times \mathbb{T}_N[X]$ is a valid TRLWE encryption of $m \in \mathcal{M} \subset \mathbb{T}_N[X]$ if $\boldsymbol{a} \xleftarrow{\$} \mathbb{T}_N[X]^k$, and $b = \langle \boldsymbol{a}, \boldsymbol{s} \rangle + m + e$, 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 practice, in this paper we only use a TRLWE encryption where $k = 1$
- **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[1]. The TRGSW encryption uses a base decomposition with we write B_g, with an exponent ℓ.

In the following, we refer to an encryption of m with the secret key $\boldsymbol{s}$ as a T(R)LWE ciphertext noted $c \in \text{T(R)LWE}_{\boldsymbol{s}}(m)$. To decrypt a sample $c \in \text{T(R)LWE}_{\boldsymbol{s}}(m)$, 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.

The ciphertext of a scalar value μ, encrypted using the noise parameter α and the key $\vec{s}$ is written $[\mu]_{\vec{s},\sigma}$. Both $\vec{s}$ and α can be omitted from the notation when their addition is not integral to the understanding of the reader and when their removal helps with clarity.

2.3 Encoding and Representation

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 any homomorphic function f, we need to create a torus-to-torus function $f_{\mathbb{T}}$ and appropriate encoding and decoding functions ι and ω.

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

Usually, these encoding and decoding functions are just re-scaling, though one must sometimes keep track of the number of operations done to ensure the output re-scaling gives the correct result.

2.4 The original bootstrapping operation

The original bootstrapping algorithm from [3] had already all the tools to implement a LUT of any negacyclic function. 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.

[1] Refer to Definition 3.6 and Lemma 3.7 in TFHE paper [6] for more information about the gadget matrix H.

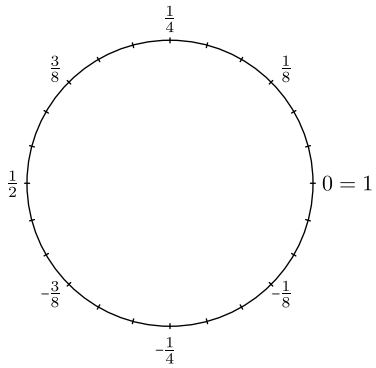

Figure 1: The representation we use for the torus and its values.

The idea behind the operation is the following. Given a $\frac{1}{2}$-antiperiodic function $f_{\mathbb{T}}$, we want to implement

$$\mathrm{LUT}\left(f_{\mathbb{T}}\right) = \left(\frac{i}{2N}, f_{\mathbb{T}}\left(\frac{i}{2N}\right)\right)_{i \in [\![-N, N-1]\!]}$$

Take an input ciphertext $[\mu]$. Take an initial polynomial containing all of the LUT outputs from positive values: the test vector

$$\mathrm{testv} = \sum_{i=0}^{N-1} f_{\mathbb{T}}\left(\frac{i}{2N}\right) X^i \in \mathbb{T}_N[X]$$

This polynomial is then privately multiplied by a re-scaling of X^{μ} through the `BlindRotate` operation. As its name suggests, this operation operates a blind rotation of the polynomial, landing us on the coefficient of the initial test vector whose index is closest to μ. We then extract the first (or another one) coefficient of the resulting rotated polynomial (the accumulator: ACC). This operation is written $\texttt{SampleExtract}_i$ for the extraction of coefficient i and we write `SampleExtract` when the coefficient extracted is 0. At this point in the original algorithm, this TLWE ciphertext is key-switched back to its original key. Here, we omit this step to gain time and reduce the output noise that the key-switching operation incurs.

Algorithm 1 shows the bootstrapping algorithm with the test vector given as input and with an extraction at coefficient 0 of the output TRLWE ciphertext ACC.

If the test vector is set to be $\mathrm{testv} = \sum_{i=0}^{N-1} \frac{1}{b} X^i$, given a base $b \in \mathbb{R}$, then the bootstrapping operation returns a LUT based on the sign function, with an output of $\frac{1}{b}$ for positive input values and $-\frac{1}{b}$ for negative input values. Figure 2

Algorithm 1 The bootstrapping algorithm.

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

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

1: Let $\bar{b} = \lfloor 2Nb \rceil$ and $\bar{a}_i = \lfloor 2Na_i \rceil \in \mathbb{Z}, \forall i \in [\![1, n]\!]$
2: ACC $\leftarrow$ `BlindRotate`(testv, $(\bar{a}_1, \ldots, \bar{a}_n, \bar{b}), (\mathsf{BK}_1, \ldots, \mathsf{BK}_n)$)
3: $\boldsymbol{c'} =$ `SampleExtract`(ACC)

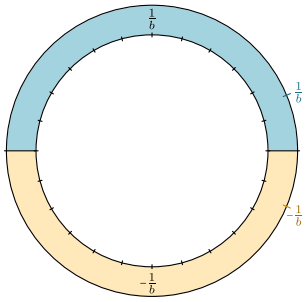

Figure 2: The representation we use for the sign bootstrapping operation. Every value in the upper (resp. lower) part of the torus is given a sign of $\frac{1}{b}$ (resp. $-\frac{1}{b}$).

shows the representation that we use for this *sign bootstrapping* operation. This can work because the sign function is negacyclic. This operation is well known and was first used in [2] for use in a private neural network evaluation.

2.5 Partial domain functional bootstrapping

The term *functional bootstrapping* (or *programmable bootstrapping*) is used to define versions of the bootstrapping algorithm that can implement LUTs based on functions that are *not* negacyclic. [7] presents most current forms that functional bootstrapping can take and present their own. Among those, we present here the *partial domain functional bootstrapping*.

Essentially, given a function $f : \mathcal{I} \to \mathcal{O}$, instead of encoding it as $f_{\mathbb{T}} : \mathbb{T} \to \mathbb{T}$, we choose ι and ω such that we encode it as $f_{\mathbb{T}^+} : [0, \frac{1}{2}[\to \mathbb{T}$. We therefore encode the plaintext space in a space half as big as it was before. However this allows us to call Algorithm 1 with a test vector

$$\text{testv} = \sum_{i=0}^{N-1} f_{\mathbb{T}^+}\left(\frac{i}{2N}\right) X^i \in \mathbb{T}_N[X]$$

Since the input μ can only be positive the output will be an encryption of the closest LUT value to $f_{\mathbb{T}^+}(\mu)$.

2.6 Private functional bootstrapping

The bootstrapping algorithm can be adapted to compute an encrypted negacyclic function. Given a function $f_{\mathbb{T}} : \mathbb{T} \to \mathbb{T}$, we create $[P_f]^{(r)}$, a TRLWE ciphertext whose i^{th} coefficient is a TLWE ciphertext encrypting $f_{\mathbb{T}}\left(\frac{i}{2N}\right)$. Such a ciphertext can either be encrypted by the secret key owner or be created by anyone with the appropriate key-switching key using the TFHE public functional key-switching operation (see Algorithm 2 of [6]) from N TLWE ciphertexts $f_{\mathbb{T}}\left(\frac{i}{2N}\right)$. We call Algorithm 1 and replace the test vector with $[P_f]^{(r)}$.

This works because the original first step of the blind rotation is a multiplication of the clear test vector with a TRGSW sample BK_1. We can replace this operation with an *external multiplication* between a TRLWE sample ($[P_f]^{(r)}$) and a TRGSW sample BK_1. For more detail on this operation and how it works, see Definition 3.12 of [6].

The method presented in Section 2.5 can be applied in the case of a private functional bootstrapping operation since it only requires a different encoding of f into $f_{\mathbb{T}^+}$. This allows us to compute this private bootstrapping over half the torus and evaluate an LUT based on any private function (negacyclic or not).

2.7 Public functional key-switching

Algorithm 2 of [6] presents their *public functional key-switching* operation which is a generalization of their original key-switching algorithm (Algorithm 2 from [3]). This operation takes p TLWE ciphertexts as inputs and builds a single TRLWE ciphertext as an output with a different key and applying a public Lipschitz morphism $f : \mathbb{T}^p \to \mathbb{T}_N[X]$ of $\mathbb{Z}$-modules. Essentially, given p ciphertexts $[x_1]_s, \ldots, [x_p]_s$, the operation outputs $[f(x_1, \ldots, x_p)]^{(r)}_{s'}$ with s a TLWE key and s' a TRLWE key not built from s. The key-switching key is a TRLWE encryption of a decomposition of s in base base and exponent t of the key s'.

Although the algorithm can be run with a host of different functions f, we present here the only version that we use in the paper for simplicity sake. The function is defined by:

$$f_p : (x_1, \ldots, x_p) \mapsto \sum_{i=0}^{p-1} X^{i\frac{N}{p}} \cdot \sum_{j=0}^{\frac{N}{p}-1} x_i X^j$$

Essentially, this function takes the p inputs and fills a size N polynomial with $\frac{N}{p}$ of each of them in order.

Optimization. The method described in [6] is not efficiently optimized when the number of inputs is lower than N. Because in this paper we only have a maximum of 4 inputs we can create a specific, pre-computed, key-switching key that drastically increases the efficiency of the key switching. This method is the same as the one presented by [11] in their Algorithm 7, and which they call "Base-aware TLWE-to-TRLWE Public Functional Key Switching". In their case, they use it for digit decomposition (hence the name) but it works exactly the same for any number of inputs that are not digits. The key-switching process is much faster this way but at the cost of a key-switching key which is as many times bigger as there are inputs (for us 4 times bigger). We refer to their algorithm for more information.

2.8 Noise propagation through homomorphic computations

As mentioned before, any TFHE encryption (whatever the encryption type) introduces an error in the ciphertext. This is done to maintain the security of the scheme but renders any computation inherently probabilistic. Given an input ciphertext with an error following a Gaussian distribution with standard deviation σ and a variance $\vartheta = \sigma^2$ [2]. In this section, we present the formulas for the variance of the noise in the output ciphertext for all the operations that interest us in the paper.

As previous articles do, we introduce $\beta = B_g/2$ and $\epsilon = \frac{1}{2B_g^\ell}$ for the expression of the noise formula.

Operation	Noise
$\texttt{Boot}_{\texttt{public}}$	$2Nn\ell\beta^2 \times \vartheta_{\mathsf{BK}} + n(N+1)\epsilon^2$
$\texttt{Boot}_{\texttt{private}}$	$\vartheta_v + 2Nn\ell\beta^2 \times \vartheta_{\mathsf{BK}} + n(N+1)\epsilon^2$
$\texttt{KeySwitch}_{f_4}$	$\vartheta_c + ntN\vartheta_{\mathsf{KS}} + \frac{1}{12}n\mathsf{base}^{-2(t+1)}$

Table 3: Operations in TFHE and their noise overhead Here ϑ_{BK} (resp. ϑ_{KS}) is the variance of the noise introduced at encryption time for the bootstrapping key (resp. the key-switching key). v is the encrypted test vector in the case of the $\texttt{Boot}_{\texttt{private}}$ operation. ϑ_c is the variance of the input TLWE ciphertexts for the $\texttt{KeySwitch}_{f_4}$ operation: we assume they all have the same and that assertion holds in our application.

Most noise formulas presented in Table 3 are taken from [6]. $\texttt{Boot}_{\texttt{public}}$ represents any public bootstrapping operation (which all have the same variance

[2] In some cases the error is described by the *width parameter*, usually written α which is defined as $\sqrt{2\pi}\sigma$

overhead). See their Algorithm 9 for reference. $\texttt{Boot}_{\texttt{private}}$ means calling `BlindRotate` with an encrypted test vector and therefore adding the variance of the test vector at the output: see their Algorithm 4. The formula for the output noise of both the TLWE-TLWE key-switch and the TLWE-TRLWE public functional key-switch is detailed in Section 3.3 of [11] where they make the bound tighter than in [6]. See their work for more precision.

2.9 FHE distance

There are situations where one needs to build a privacy-preserving 1 Nearest Neighbour (1-NN) operator. Our min/argmin algorithms are a good solution for this provided we can compute a distance homomorphically. This can be done either between two encrypted vectors or an encrypted vector and a clear domain vector. Examples of this use can be found in previous articles, see for instance [20, 21].

3 Our FHE min/argmin algorithms

In this section, we present the algorithms that we have developed to obtain a fast and accurate min/argmin operation. Sections 3.3 and 3.2 present two variants on what we call a *tournament method*, one quicker than the other at the expense of some accuracy. In Section 3.4, we provide an in-depth analysis of the accuracy and precision of our algorithms.

3.1 Tournament Method

As the name suggest, this method computes the overall minimum among a collection of arguments by constructing a tree. It computes the minimum between two arguments at every level with the minimum values populating the next level of nodes in the tree. If one wishes to only compute the minimum overall value, one could omit populating the next level of the tree with the indices of the minimum values from the previous level. The algorithms that we present in this section allow for both a min computation and an argmin computation at no additional performance cost. These algorithms can be adapted to compute a max/argmax operation trivially.

In the following, given two encrypted values $[x_i]$ and $[x_j]$ and their encrypted indices $[i]$, $[j]$, we build fast and accurate homomorphic computations of $[\min(x_i, x_j)]$ and $[\text{argmin}(x_i, x_j)]$. From those, the next level of the tree can be iterated. We assume that $x_i, x_j \in \mathbb{T}$ and do not bother here with considerations about encoding.

In this paper, we present two novel ways to do this. The first one with a sign computation to create a selector value before a MUX (multiplexor) gate is applied. A second one where the difference of the two values $[x_i]$ and $[x_j]$ is directly used as a selector value for the MUX gate.

3.2 Tournament with sign selection

Difference. The first step is to obtain the difference of the two inputs with an homomorphic subtraction: $[x_i] - [x_j] = [x_i - x_j]$. This is a trivial operation that "only" doubles the noise in the output ciphertext with respect to the initial noise. However, for this to make any sense, we need to have $x_i, x_j \in [-\frac{1}{4}, \frac{1}{4}]$. Therefore $x_i - x_j \in \mathbb{T}$.

Sign. We mention in Section 2.4 that it is easy to obtain a sign function output from the original bootstrapping operation. Indeed, the sign function is negacyclic and this has been known for some time in the community. If the test vector in Algorithm 1 is set to testv $= \sum_{i=0}^{N-1} \frac{1}{16} X^i$ we obtain a bootstrapping operation we call $\texttt{Boot}_{\texttt{sign}}$. Its output is:

$$[s_{i,j}] = \texttt{Boot}_{\texttt{sign}}\left([x_i - x_j] - \frac{1}{4N}\right) = \begin{cases} \left[\frac{1}{16}\right] & \text{if } x_i > x_j \\ \left[-\frac{1}{16}\right] & \text{if } x_i < x_j \end{cases}$$

with the notation $s_{i,j}$ used to simplify later expressions.

Bootstrapping a 0 value. Here, we need to address the behavior of the bootstrapping operation around 0. Because it is a LUT and not a continuous function, if $x_i - x_j = 0$, then the output of $\texttt{Boot}_{\texttt{sign}}$ will be $\left[\frac{1}{16}\right]$ 100% of the time. We want it to have a 50% chance to output either $\left[\frac{1}{16}\right]$ or $\left[-\frac{1}{16}\right]$ for an input value of [0]. For this, the input needs to be "rotated" by $\frac{1}{4N}$: $[x_i] - [x_j] - \frac{1}{4N}$; where N is the size of the bootstrapping key as seen in Section 2. Indeed, after the rotation applied here and the re-scaling by $2N$ in the bootstrapping algorithm (Algorithm 1), an input value of $x_i - x_j = 0$ would therefore be equal to $\lfloor -\frac{1}{2} \rceil$. This is -1 or 0 with equal probability, and hence an output of $-\frac{1}{16}$ or $\frac{1}{16}$ with equal probability.

Min and argmin selection. Using the partial domain functional bootstrapping method presented in Section 2.5, we could imagine a way to select the min and argmin values of $[x_i]$ and $[x_j]$. We set the test vector from the original bootstrapping algorithm (Algorithm 1) to:

$$\text{testv} = \sum_{l=0}^{\frac{N}{4}-1} x_i X^l + \sum_{l=\frac{N}{4}}^{\frac{N}{2}-1} x_j X^l + \sum_{l=\frac{N}{2}}^{\frac{3N}{4}-1} \frac{i}{b} X^l + \sum_{l=\frac{3N}{4}}^{N-1} \frac{j}{b} X^l \tag{1}$$

given a base b with which to encode the indices i and j (making sure that $\frac{i}{b}, \frac{j}{b} \in \mathbb{T}$). We apply Algorithm 1 without the last `SampleExtract` step. Therefore we output a TRLWE encryption of a degree $N-1$ polynomial. We call this operation as $\texttt{Boot}_{\texttt{select}}$. For a random input of $\omega \in [0, \frac{1}{4}[$, the output of the operation is

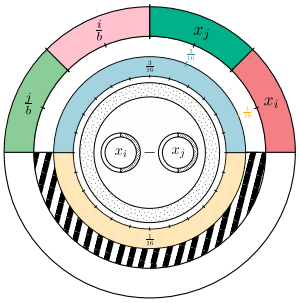

Figure 3: The inner torus represents the sign bootstrapping operation as in Figure 2 though with the output rotated by $\frac{1}{8}$. The outer torus represents the private functional bootstrapping operation given the test vector in Equation 1. Because our functional bootstrapping method is the partial domain one, the lower part of the torus cannot be used as input. Dotted spaces in the torus represent all possible values at a given time.

$$\sum_{l=0}^{\frac{N}{8}-1} x_i X^l + \sum_{l=\frac{N}{8}}^{\frac{3N}{8}-1} x_j X^l + \sum_{l=\frac{3N}{8}}^{\frac{5N}{8}-1} \frac{i}{b} X^l + \sum_{l=\frac{5N}{8}}^{\frac{7N}{8}-1} \frac{j}{b} X^l - \sum_{l=\frac{7N}{8}}^{N-1} x_i X^l \tag{2}$$

when $\omega \in [0, \frac{1}{8}[$ and

$$\sum_{l=0}^{\frac{N}{8}-1} x_j X^l + \sum_{l=\frac{N}{8}}^{\frac{3N}{8}-1} \frac{i}{b} X^l + \sum_{l=\frac{3N}{8}}^{\frac{5N}{8}-1} \frac{j}{b} X^l - \sum_{l=\frac{5N}{8}}^{\frac{7N}{8}-1} x_i X^l - \sum_{l=\frac{7N}{8}}^{N-1} x_j X^l \tag{3}$$

when $\omega \in [\frac{1}{8}, \frac{1}{4}[$.

Therefore:

$$\texttt{SampleExtract}_0 \left(\texttt{Boot}_{\texttt{select}} \left([s_{i,j}] + \frac{1}{8} - \frac{1}{4N} \right) \right) = \begin{cases} [x_j] & \text{if } x_i > x_j \\ [x_i] & \text{if } x_i < x_j \end{cases}$$

and

$$\texttt{SampleExtract}_{\frac{N}{2}} \left(\texttt{Boot}_{\texttt{select}} \left([s_{i,j}] + \frac{1}{8} - \frac{1}{4N} \right) \right) = \begin{cases} [\frac{j}{b}] & \text{if } x_i > x_j \\ [\frac{i}{b}] & \text{if } x_i < x_j \end{cases}$$

The addition of $\frac{1}{8}$ after the sign computation is to rescale the possible output values $\left\{-\frac{1}{16}, \frac{1}{16}\right\}$ up to $\left\{\frac{1}{16}, \frac{3}{16}\right\}$. The addition of $\frac{-1}{4N}$ is for the same reason as for the sign bootstrapping operation.

This solves our problem perfectly except for the fact that we need clear values x_i and x_j in order to build the test vector in Equation 1, thus defeating the purpose of FHE altogether. Thankfully, as presented in Section 2.6, we can obtain the same result if we use an encrypted test vector instead, using the private functional bootstrapping method.

Encrypted test vector creation. In order to create a TRLWE encryption of the test vector in Equation 1, we use the public functional key-switching method presented in Section 2.7. In keeping with the notation of that section, we use the function f_4 as the public function of our key-switching and call the public functional key-switching operation with inputs $[x_i]$, $[x_j]$, $\left[\frac{i}{b}\right]$ and $\left[\frac{j}{b}\right]$. By definition, we obtain the exact test vector in Equation 1 in encrypted form.

3.3 Tournament without sign selection

The use of a sign bootstrapping operation is not necessary in some cases. Indeed, the output of the $\texttt{Boot}_{\texttt{select}}$ operation as presented in Equations 2 and 3 is the same for all inputs respectively in $[0, \frac{1}{8}]$ and $[\frac{1}{8}, \frac{1}{4}]$ and not just for $\frac{1}{16}$ and $\frac{3}{16}$. Therefore, if one takes x_i and x_j to be in $[-\frac{1}{16}, \frac{1}{16}]$ (and not $[-\frac{1}{4}, \frac{1}{4}]$), then we have $x_i - x_j \in [-\frac{1}{8}, \frac{1}{8}]$ and

$$x_i - x_j + \frac{1}{8} \in \begin{cases} [\frac{1}{8}, \frac{1}{4}] & \text{if } x_i > x_j \\ [0, \frac{1}{8}] & \text{if } x_i < x_j \end{cases}$$

Therefore, we can use $\left[x_i - x_j + \frac{1}{8} - \frac{1}{4N}\right]$ as a direct input to the $\texttt{Boot}_{\texttt{select}}$ operation. This has the benefit of removing a bootstrapping operation entirely. Given the weight that the bootstrapping operations have in the running time of the algorithm, this is very significant. More details are given in Section 4. Removing the sign bootstrapping operation has two drawbacks :

- Because the $\texttt{Boot}_{\texttt{select}}$ operation is a partial domain functional bootstrapping operation, we have a margin for error that is halved compared with a full domain bootstrapping operation such as $\texttt{Boot}_{\texttt{sign}}$. Indeed, if the homomorphic LUT has N slots here compared with $2N$ for the sign.
- On top of this, the encoding of the data to compare (x_i and x_j) makes them four times smaller. However the noise that we need to introduce at encryption time remains the same. Therefore the initial ciphertext noise is four times bigger relative to the actual (not rescaled for encoding) data.

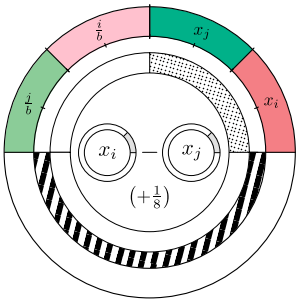

Figure 4: The functional bootstrapping operation representation given $x_i - x_j + \frac{1}{8}$ as input and no sign bootstrapping. Dotted spaces in the torus represent all possible values at a given time.

3.4 Noise Analysis

Section 2.8 presents the formulas for the variance overhead of every operation that we use in our FHE algorithm. In this section we use theses formulas to analyse the precision of our algorithms. When using this term - *precision* - for a given bootstrapping operation, there are two distinct phenomenons that this could actually point to:

- the error in the input ciphertext to a given bootstrapping operation does not influence the output error as seen in Table 3. This is the whole point of bootstrapping. However it may affect the actual output (and not its error). Say for instance that we have an input to a $\texttt{Boot}_{\texttt{sign}}$ operation: a ciphertext of a negative value $[m + e]$ with $m < 0$ but with an error e such that $m + e > 0$. Then the output might be $\left[\frac{1}{16} + e'\right]$ with error e' decorrelated from e. In the following, we talk about the *accuracy* of the operation.
- as for the output error, it is controlled both by the parameters of the bootstrapping key (ϑ_{BK}, N, B_g and ℓ) and (if it is a private boot operation) the error in the encrypted test vector. That error needs to be analysed so that it does not grow to an extent that the output cannot be used for further computations. In the following, we talk about the *precision* of the operation.

The *precision* of a given homomorphic application can be easily evaluated by giving an upper bound on the the variance of the output ciphertext. This is what we do in Section 3.4.1. The *accuracy* of an homomorphic application is a harder notion to define formally. For this reason we introduce Definition 1. For simplicity sake, we set the type of plaintext space to be any space $\mathcal{M}$ of odd size $M \in \mathbb{N}$ such that $\mathcal{M} = \left\{-\frac{M-1}{2}, \ldots, 0, \ldots, \frac{M-1}{2}\right\} \subset \mathbb{N}$. Our accuracy results, presented in Section 3.4.2, are only valid for this kind of plaintext space but one can easily see how they can be adapted depending on the use-case.

Definition 1 (Worst Case Accuracy)**.** *The worst case accuracy of a bootstrapping operation is defined with respect to the input plaintext space $\mathcal{M}$ of size $M \in \mathbb{N}$ and a probability $p \in [0,1]$. Given a plaintext space size M, we say that a bootstrapping operation is accurate in the worst case with probability p if, for every plaintext member taken as input, the probability of a correct result is more than p.*

Essentially, by setting a probability considered as "overwhelming", Definition 1 allows us to determine a plaintext space size for which all computations are accurate when all of them are at the most risk of mistake. In our case, that means the eventual winning value μ of the argmin/min tournament is compared, at every step, with $\mu + 1$.

3.4.1 Precision of the algorithm

First of all, we define layers of our tournament tree as going from 1 to L with each layer corresponding to a comparison step in the tournament. Therefore 2^L comparisons require L layers.

In this section we evaluate the precision of the algorithm in the two different variations of our protocol. An evaluation of the output noise of this operation at the last layer of the tournament tree yields an upper bound on the noise in our $[x_i]$ ciphertexts throughout all the other layers of the tree. We call $\vartheta^{(l)}$ the variance (actually an upper bound) of the noise of the $[x_i]$ ciphertext at the l^{th} layer of the tournament tree[3], before the comparison step. Therefore $\vartheta^{(1)}$ is the variance of the ciphertexts before the algorithm is applied. $\sigma^{(l)}$ is its corresponding standard deviation.

with sign In the case where we use a sign bootstrapping first, we need to introduce a simple TLWE-TLWE key-switch after $\texttt{Boot}_{\texttt{sign}}$, but the output noise is not affected by it.

The output variance overhead at layer $l+1$ with regard to that at layer l can be written:

$$\vartheta^{(l+1)} \leq \vartheta^{(l)} + ntN\vartheta_{\mathsf{KS}} + n\frac{1}{12}\mathsf{base}^{-2(t+1)} + 2Nn\ell\beta^2\vartheta_{\mathsf{BK}} + n(N+1)\epsilon^2$$

Therefore, by recurrence, with regard to the first layer of the tree, we can write:

$$\vartheta^{(l)} \leq \vartheta^{(1)} + l \times \left(ntN\vartheta_{\mathsf{KS}} + n\frac{1}{12}\mathsf{base}^{-2(t+1)} + 2Nn\ell\beta^2\vartheta_{\mathsf{BK}} + n(N+1)\epsilon^2 \right)$$

[3]not to be confused with ℓ, the TFHE parameter

no sign. This variance overhead is also the one for the case where we omit the sign bootstrapping operation and simply run the $\texttt{Boot}_{\texttt{select}}$ operation on the difference of the two values (Section 3.3). However, in that case, the values are encoded over $[-\frac{1}{16}, \frac{1}{16}]$ instead of $[-\frac{1}{4}, \frac{1}{4}]$ for the version with $\texttt{Boot}_{\texttt{sign}}$ (Section 3.2). Therefore, even though the variance of the result is the same, the relative error is 4 *times as high* when not using a sign bootstrapping operation.

3.4.2 Accuracy of the algorithm

In this section we measure the worst-case accuracy of both versions of our algorithm in accordance with Definition 1. Specifically, we make explicit the link between the parameters of the scheme (B_g, ℓ, ...), the target probability of success p and the plaintext size M.

We first analyse the accuracy of a single $\texttt{Boot}_{\texttt{sign}}$ operation and the following $\texttt{Boot}_{\texttt{select}}$ operation in order to infer the overall worst-case accuracy of the algorithm with sign. Then we do the same for a single $\texttt{Boot}_{\texttt{select}}$ operation and infer the overall accuracy of the algorithm without sign.

First, for a given probability of success $p \in [0, 1]$ and a given standard deviation σ, we define the maximum possible error $e_{\max}(p, \vartheta) > 0$ as the smallest value such that:

$$\frac{1}{\sqrt{2\pi\vartheta}} \int_{-e_{\max}(p,\vartheta)}^{e_{\max}(p,\vartheta)} f_{\mathcal{N}}\left(t|0,\vartheta\right)\, dt \geq p$$

where $f_{\mathcal{N}}\left(x|\mu,\vartheta\right)$ is the density function for the general Normal distribution.

Accuracy of $\texttt{Boot}$ operations. As [11] explain in detail in their Section 3.3.1, the noise added by the rounding step of the bootstrapping algorithm corresponds to the sum of n uniformly distributed variables each with variance $\frac{1}{12} \cdot \left(\frac{1}{2N}\right)^2$. This results in an Irwin-Hall distributed variable of variance $\frac{N}{12} \cdot \left(\frac{1}{2N}\right)^2$ which, as they do, we assimilate to a normally distributed variable. That noise is additive with the noise initially present in the ciphertext. Therefore at level l of the tournament tree, the input noise of the $\texttt{Boot}_{\texttt{sign}}$ operation (or $\texttt{Boot}_{\texttt{select}}$ in the case there is no sign computation) has a variance of:

$$\vartheta_b^{(l)} = 2\vartheta^{(l)} + \frac{1}{48N}$$

$\vartheta_b^{(l)}$ is the "real" variance at the input of a bootstrapping operation at layer l. ϑ_b is the variance at the output of a bootstrapping operation at any level (by definition it is constant).

We define r such that input values (after re-scaling) are in the interval $\left[-\frac{1}{r}, \frac{1}{r}\right]$. When $\texttt{Boot}_{\texttt{sign}}$ is used, we have $r = 4$, and $r = 16$ otherwise as seen in Section

3.1. Because of how our plaintext space was defined, at level l of our tournament tree, the maximum plaintext space size $M_{\max}$ for which the result of a bootstrapping operation is accurate in the worst case with probability p according to definition 1 is:

$$M_{\max} = \left\lfloor \frac{2}{r \times e_{\max}\left(p, \vartheta_b^{(l)}\right)} \right\rfloor \tag{4}$$

Accuracy of the overall algorithm. Here we talk about overall worst-case accuracy as a straight-forward extension of Definition 1 to our algorithm as a whole. The probability therefore is that of the final result being the actual min/argmin.

Theorem 1 (Algorithm with sign Accuracy)**.** *Given two probability values p_{sign} and p_{select}, for our algorithm with sign to be worst-case accurate over all with a plaintext size of M, a number of comparisons $< 2^L$, and probability at least $p_{sign}^L \times p_{select}^L$ it is sufficient that:*

$$M \leq \frac{2}{4 \times e_{\max}\left(p_{sign}, \vartheta_b^{(L)}\right)} \tag{5}$$

and

$$4 \leq \frac{2}{16 \times e_{\max}\left(p_{select}, \vartheta_b + \frac{1}{48N}\right)} \tag{6}$$

where ϑ_b is the upper bound on the variance at the output of the sign bootstrapping operation as given in Table 3.

Proof. The first condition (Equation 5) is an expression of the condition that, at the last layer of the tournament, the $\texttt{Boot}_{\texttt{sign}}$ operation must be worst-case accurate with probability p. If that holds, because $\forall l < L, \sigma^{(l)} < \sigma^{(L)}$, that means all other previous $\texttt{Boot}_{\texttt{sign}}$ operations were worst-case accurate with probability at least $p_{\texttt{sign}}$. This makes the probability that all of the $\texttt{Boot}_{\texttt{sign}}$ operations in the tournament are worst-case accurate at least $p_{\texttt{sign}}^{2^L-1}$. However, we don't need every comparison to be accurate, just the ones that involve the eventual winner. That means only L $\texttt{Boot}_{\texttt{sign}}$ operations must be accurate. Therefore, the probability goes up to $p_{\texttt{sign}}^L$

The second condition (Equation 6) is an expression of the condition that, at any level of the tournament, the $\texttt{Boot}_{\texttt{select}}$ operation be worst-case accurate with probability $p_{\texttt{select}}$. This is true because the output of the $\texttt{Boot}_{\texttt{sign}}$ operation effectively has a plaintext size of 4. Therefore we can apply our accuracy result

in Equation 4 with $r = 16$ and $M = 4$. Then, as before, we need only L operations to be correct. That leads to a probability of $p^L_{\texttt{select}}$ overall. □

Theorem 2 (Algorithm without sign Accuracy)**.** *For our algorithm without sign to be worst-case accurate over all with a plaintext size of M, a number of comparisons $< 2^L$, and probability at least p^L it is sufficient that:*

$$M \leq \frac{2}{16 \times e_{\max}\left(p, \vartheta_b^{(L)}\right)} \tag{7}$$

Where $\vartheta^{(L)}$ is the upper bound on the standard deviation in the ciphertexts before the last comparison step as defined in Section 3.4.1.

Proof. The condition in Equation 7) is an expression of the condition that, at the last layer of the tournament, the $\texttt{Boot}_{\texttt{select}}$ operation must be worst-case accurate with probability p, as per Equation 4 when $r = 16$. □

On the error-resilience of such an algorithm All of the evaluations above strive to provide a strict mathematical structure in which one can evaluate our homomorphic algorithm on its ability to provide exactly the same results as a clear-domain algorithm would. This would obviously be great to achieve in all cases. Sadly HE is not at that point yet. A consolation can be found in the fact that, in a lot of cases, a small computational error can be either invisible or not too damaging too many real-case scenarios. Indeed, our algorithm is built in such a way that errors will occur when two values are close to each-other. Many applications require the computation of an argmin over values that are all very high except for one because such was the goal of the pre-processing done beforehand (see embedding-based neural networks [20] or collaborative learning approaches [17]).

4 Performance and Experimental results

Our two levelled homomorphic algorithms are evaluated in this section, both for their time performance and their precision. Parameter choices are explained, and we strive to present the main options at the disposal of a user of these algorithms. All times were obtained on an Intel Core i7-6600U CPU @ 2.60GHz, with *no multi-threading*.

4.0.1 Parameter selection

For this algorithm, we use three main homomorphic operations, all with their own parameter set. These parameters determine both the variance upper bound for the output noise and the computation time for the operation. All of this comes with the usual time/precision trade-off. There are a vast number of

$\log_2(B_g)$	ℓ	time (ms)	$\vartheta_{\text{boot}} \times 10^9$
1	26	246	3.5
2	13	129	7.0
2	12	115	7.3
3	8	82	18
4	6	65	52
5	5	60	170
6	4	53	380

Table 4: The possible parameter choices for a bootstrapping that we recommend. These are chosen carefully. Any combination of parameters that have a strictly better alternative both in terms of time and variance are not included. The other parameters are set to $\alpha = 2 \cdot 10^{-8}$ and $N = 1024$ for a security level of $\lambda = 120$.

$\log_2(\mathsf{base})$	t	time (ms)	$\vartheta_{\text{boot}} \times 10^9$
3	10	92	3.4
5	6	55	14
6	5	47	40
7	4	38	300

Table 5: The possible parameter choices for a public functional key-switching operation that we recommend. These are chosen carefully. Any combination of parameters that have a strictly better alternative both in terms of time and variance are not included. The other parameters are set to $\alpha = 2 \cdot 10^{-8}$ and $N = 1024$ for a security level of $\lambda = 120$.

possible combinations for parameter choices and to help the reader with their own choice depending on their constraints we present in Table 4 (resp. Table 5) the possible choices that we recommend for the bootstrapping parameters B_g and ℓ (resp. the key-switching parameters base and t). Times are measured experimentally and variance upper bounds are computed using the formulas given in Table 3.

The initial ciphertexts are encrypted using a key size $n = 1024$ and a noise parameter $\alpha = 2 \cdot 10^{-8}$ (the standard deviation σ corresponds to $\sigma = \frac{\alpha}{\sqrt{2\pi}}$). This, according to the latest work on LWE-based encryption schemes, ensures a security level of $\lambda = 120$. We use for this the latest commit of the LWE estimator[4] [1, 15]. The same values (key size and noise parameter) are used for all encryptions of the evaluation keys (bootstrapping keys and key-switching keys). While some tweaking could be used to optimize the noise output and the computation time, we did not look into it.

[4]https://bitbucket.org/malb/lwe-estimator/

On TFHE's implementation – The parameters that we present in tables 4 and 5 are all constrained by the implementation of TFHE. In the implementation that we use, torus values are written over 32 bits. This means we are constrained by $\ell \times \log_2(B_g) < 32$ and $t \times \log_2(\mathsf{base}) < 32$ for both decomposition algorithms to provide a correct result (otherwise the values overflow). This limits the precision we can attain using the public functional key-switching operation (though it does not limit the precision of the bootstrapping) which could go down to a variance of $3 \cdot 10^{-10}$. A 64-bit implementation does exist in the original TFHE library that we use but is not as optimized as the 32-bit version, thus our choice.

4.1 Theoretical analysis

Here we present precision, accuracy and time performance results for our algorithms in an abstract setting (no real-world application) to give a general overview of their performance.

As in Section 3.4, we differentiate between accuracy and precision. We compute the maximum size (in bits) of a plaintext space for which our algorithm is accurate in the worst case with probability $p > 1 - 2^{-32}$ (at most one error for every 2^{32} runs of the argmin computation) according to Theorems 1 and 2. We call this value *acc*. Given the noise formulas presented in Section 3.4.1, we can define *prec* to be the maximum plaintext size (in bits) for which ciphertexts can be decrypted correctly throughout the computation (including the overall result) with probability $p > 1 - 2^{-32}$.

Table 6 presents *acc* and *prec* values for two sets of parameters and for both our algorithm *with sign* and *without sign*. The "slow" parameter corresponds to $B_g = 2^1$, $\ell = 26$, $\mathsf{base} = 2^3$, $t = 10$. The "meh" parameter corresponds to $B_g = 2^4$, $\ell = 6$, $\mathsf{base} = 2^5$, $t = 6$. The "fast" one corresponds to $B_g = 2^6$, $\ell = 4$, $\mathsf{base} = 2^7$, $t = 4$. We limit our table to those two sets for the sake of clarity but those results can be obtained for any set of parameters using the formulas provided in this paper.

4.2 Argmax computation in the PATE framework

The PATE framework, presented in [14] and summarized in Section 1.1, requires the computation of an argmax over encrypted values. In order to test the performance of our algorithm we ran it on the initial datasets used by [14]. They had teachers vote on labels for both the MNIST dataset and the SVHN dataset[5]. There are 250 teachers and therefore the votes can go from 0 (if nobody votes for a label) to 250 (if everybody agrees on a label). In both MNIST and SVHN, there are 10 classes and therefore we need to find the argmax among 10 votes for each aggregation.

[5] we took the teacher's votes from `https://github.com/npapernot/multiple-teachers-for-privacy`

speed	L	2^L	sign			no sign		
			acc	*prec*	time	*acc*	*prec*	time
slow..	1	2	4	9	0.58	2	7	0.39
	7	128	4	8	79.16	2	6	42.63
	10	1024	4	8	643.82	2	6	343.5
meh.	1	2	4	8	0.18	2	6	0.11
	7	128	4	6	24.15	2	4	15.7
	10	1024	4	6	192.4	2	4	113.5
fast!	1	2	4	6	0.13	2	4	0.12
	7	128	4	5	17.02	2	2	10.3
	10	1024	4	4	134.69	2	2	81.25

Table 6: A table comparing the two versions of our algorithm: with sign and without. L indicates the depth of the tournament tree and therefore 2^L is the number of values compared with that tree. *acc* (in bits) is the plaintext space size that ensures worst-case accuracy (Definition 1) with probability $p > 1 - 2^{-32}$. *prec* (in bits) is the plaintext space size at the output of the algorithm with probability $p > 1 - 2^{-32}$. The timings, which are presented in "seconds", are the experimental times required for computing the argmin using the respective algorithms.

The parameters are as follows: $N = 1024$, $\alpha = 2 \cdot 10^{-8}$, $l = 6$, $\log_2(B_g) = 4$, $t = 6$ and $\log_2(\mathsf{base}) = 5$. These provide a security level of $\lambda = 120$. Over the MNIST votes, our approach has an accuracy of 99.96% with the correct argmax being computed for 8996 data point out of 9000. Over the SVHN votes, our argmax yields 99.92% accuracy over 26032 data points (20 mistakes). All the mistakes happen when there are two votes that are closer than 4. In both cases, the argmax among the 10 labels took 1.65s.

5 Conclusion

Our work introduces a novel method to compare a number of encrypted values efficiently. Though it is less precise than previous methods based on bitwise decomposition for instance, real-world applications show that it can provide results almost as good as in the clear domain, with a significant decrease in computation time when not comparing several sets of values in parallel. Such a private comparison operator is the base for a wide variety of very useful machine learning and statistical analysis tools. Building any of those tools in privacy-preserving manner becomes much easier with the use of our work.

References

[1] Martin R. Albrecht, Rachel Player, and Sam Scott. "On the concrete hardness of Learning with Errors". In: *Journal of Mathematical Cryptology* 9.3 (2015), pp. 169–203. DOI: doi:10.1515/jmc-2015-0016. URL: https://doi.org/10.1515/jmc-2015-0016.

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

[3] 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.

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

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

[6] 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.

[7] Pierre-Emmanuel Clet, Martin Zuber, Aymen Boudguiga, Renaud Sirdey, and Cédric Gouy-Pailler. *Putting up the swiss army knife of homomorphic calculations by means of TFHE functional bootstrapping*. Cryptology ePrint Archive, Report 2022/149. https://ia.cr/2022/149. 2022.

[8] 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.

[9] Craig Gentry. "A Fully Homomorphic Encryption Scheme". PhD thesis. Stanford, CA, USA, 2009. ISBN: 9781109444506.

[10] Craig Gentry. "Fully Homomorphic Encryption Using Ideal Lattices". In: STOC '09 (2009).

[11] 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.

[12] Ilia Iliashenko and Vincent Zucca. "Faster homomorphic comparison operations for BGV and BFV". In: *Proceedings on Privacy Enhancing Technologies* 2021.3 (2021), pp. 246–264. DOI: 10.2478/popets-2021-0046.

[13] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Advances in Cryptology — EUROCRYPT '99*. Ed. by Jacques Stern. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1999, pp. 223–238. ISBN: 978-3-540-48910-8. DOI: 10.1007/3-540-48910-X_16.

[14] Nicolas Papernot, Shuang Song, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Úlfar Erlingsson. "Scalable Private Learning with PATE". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=rkZB1XbRZ.

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

[16] R L Rivest, L Adleman, and M L Dertouzos. "On Data Banks and Privacy Homomorphisms". In: *Foundations of Secure Computation, Academia Press* (1978), pp. 169–179.

[17] 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.

[18] Anselme Tueno and Jonas Janneck. *A Method for Securely Comparing Integers using Binary Trees*. Cryptology ePrint Archive, Report 2021/1646. https://ia.cr/2021/1646. 2021.

[19] Andrew C. Yao. "Protocols for secure computations". In: *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. 23rd Annual Symposium on Foundations of Computer Science (sfcs 1982). ISSN: 0272-5428. Nov. 1982, pp. 160–164. DOI: 10.1109/SFCS.1982.38.

[20] Martin Zuber, Sergiu Carpov, and Renaud Sirdey. "Towards Real-Time Hidden Speaker Recognition by Means of Fully Homomorphic Encryption". In: *Information and Communications Security - 22nd International Conference, ICICS 2020, Copenhagen, Denmark, August 24-26, 2020, Proceedings*. Ed. by Weizhi Meng, Dieter Gollmann, Christian Damsgaard Jensen, and Jianying Zhou. Vol. 12282. Lecture Notes in Computer Science. Springer, 2020, pp. 403–421. DOI: 10.1007/978-3-030-61078-4_23. URL: https://doi.org/10.1007/978-3-030-61078-4%5C_23.

[21] 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.