Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks

Ilaria Chillotti, Marc Joye, and Pascal Paillier

Zama, Paris, France

Abstract. In many cases, machine learning and privacy are perceived to be at odds. Privacy concerns are especially relevant when the involved data are sensitive. This paper deals with the privacy-preserving inference of deep neural networks.
We report on first experiments with a new library implementing a variant of the TFHE fully homomorphic encryption scheme. The underlying key technology is the programmable bootstrapping. It enables the homomorphic evaluation of any function of a ciphertext, with a controlled level of noise. Our results indicate for the first time that deep neural networks are now within the reach of fully homomorphic encryption. Importantly, in contrast to prior works, our framework does not necessitate re-training the model.

1 Introduction

The machine learning revolution. The basic premise of learning from data is to uncover a process from a set of observations [1,19]. Machine learning is very efficient at extracting useful information and can dynamically adapt to a certain task given the data and the desired goal. Hence, unlike classical statistical methods, it learns the important impact factors of the model from the data itself.

Machine learning enables the development of numerous applications: regression, classification, recommendation, clustering, anomaly detection, and more. It is nowadays routinely employed in various sectors and offers a multitude of services, including fraud detection, risk management, face recognition, driving assistants, email classification, automatic translation, medical diagnosis, etc.

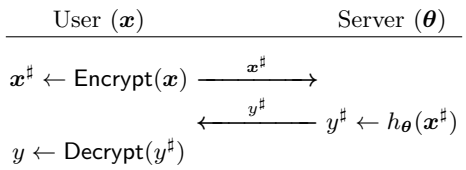
The need for privacy. Machine learning algorithms are extremely useful in many areas but the type of data that they deal with is often sensitive. Typical examples include algorithms for the detection of certain genetic diseases from DNA samples or the ones used for face recognition or email classification, to name a few. The processed data contain private information about users and could be used in many ways, from target advertising to blackmail or even threat in some cases. This is why it is essential to protect the data being used in machine learning applications.

Privacy requirements are also pushed by recent regulations companies dealing with user's data must comply with, like the GDPR (General Data Protection Regulation) [16] in Europe or the CCPA (California Consumer Privacy Act) [10] in the US.

Fully homomorphic encryption. Cryptographic techniques are methods of choice when it comes to the protection of data. But traditional encryption algorithms merely protect data while it is in transit or at rest. Indeed, one limitation and structural property of traditional encryption schemes is that data first needs to be decrypted prior to being processed. As discussed earlier, this is not suited for machine learning applications.

With traditional encryption schemes, the privacy control lies in the hands of the recipient of the encrypted data. A fundamentally different approach is to rely on *fully homomorphic encryption* (FHE), first posed as a challenge in 1978 [29] and only solved in 2009 in a breakthrough result by Gentry [17]. In contrast to traditional encryption schemes, fully homomorphic encryption schemes allow the recipient to directly operate on encrypted data.

An FHE-based approach perfectly fits the use-case of *machine-learning-as-a-service* (MLaaS). A typical application scenario is shown in Fig. 1. It involves a user sending data to a service provider (server) running a trained model for a given task (e.g., medical diagnosis).



A server offering MLaaS owns a model defined by its parameters $\boldsymbol{\theta}$. A client needs the prediction $y = h_{\boldsymbol{\theta}}(\boldsymbol{x})$ of this model for a new input data $\boldsymbol{x}$. Using FHE, everything works in the same way. The user sends an encryption $\boldsymbol{x}^\sharp$ of her data $\boldsymbol{x}$. The server evaluates the model over $\boldsymbol{x}^\sharp$, obtains $y^\sharp$, and returns it to the user. The user decrypts $y^\sharp$ and obtains the prediction y in the clear. Correctness follows from the homomorphicity of the encryption algorithm: $y^\sharp := h_{\boldsymbol{\theta}}(\boldsymbol{x}^\sharp) = h_{\boldsymbol{\theta}}(\mathsf{Encrypt}(\boldsymbol{x})) = \mathsf{Encrypt}(h_{\boldsymbol{\theta}}(\boldsymbol{x})) = \mathsf{Encrypt}(y)$ and so $\mathsf{Decrypt}(y^\sharp) = y$.

Fig. 1: MLaaS over encrypted data.

Controlling the noise. At the core of Gentry's result resides the technique of *bootstrapping*. All known instantiations of fully homomorphic encryption schemes produce noisy ciphertexts. Running homomorphic operations on these ciphertexts in turn increases the noise level in the resulting ciphertext. At some point, the noise present in a ciphertext may become too large and the ciphertext is no longer decryptable. A homomorphic encryption scheme supporting a predetermined noise threshold is termed *leveled* (or somewhat homomorphic). Bootstrapping is a generic technique that allows refreshing ciphertexts. It therefore enables one to turn leveled homomorphic encryption schemes into *fully* homomorphic encryption schemes, and so to make them evaluate any possible function on ciphertexts. The key idea behind bootstrapping is to homomorphically evaluate the decryption circuit. An encryption of the decryption key (matching the encryption key used to produce the ciphertext) has to be provided in order to perform the bootstrapping procedure and is called bootstrapping key. This is detailed in Fig. 2.

$$\mathsf{Encrypt}_{\mathsf{sk}}(x) \rightarrow \boxed{f(\cdot)} \rightarrow \mathsf{Encrypt}_{\mathsf{sk}}(f(x))$$

(a) For a (symmetric) FHE algorithm $\mathsf{Encrypt}$, given the encryption of x under private key sk, the homomorphic evaluation of a univariate function f yields the encryption of $f(x)$.

$$\mathsf{Encrypt}_{\mathsf{sk}_2}(\mathsf{sk}_1) \rightarrow \boxed{\mathsf{Decrypt}(\cdot, c)} \rightarrow \mathsf{Encrypt}_{\mathsf{sk}_2}(\mathsf{Decrypt}(\mathsf{sk}_1, c)) = \mathsf{Encrypt}_{\mathsf{sk}_2}(m)$$

with $c \leftarrow \mathsf{Encrypt}_{\mathsf{sk}_1}(m)$

(b) Let $c \leftarrow \mathsf{Encrypt}_{\mathsf{sk}_1}(m)$ be a [very] noisy ciphertext encrypting a plaintext m and let $\mathsf{bk} \leftarrow \mathsf{Encrypt}_{\mathsf{sk}_2}(\mathsf{sk}_1)$ denote the bootstrapping key. Assume that f is the decryption function dedicated to ciphertext c, viewed as the univariate function $\mathsf{Decrypt}(\cdot, c)$. Then, letting $x = \mathsf{sk}_1$, the homomorphic evaluation of f yields $\mathsf{Encrypt}_{\mathsf{sk}_2}(f(x)) = \mathsf{Encrypt}_{\mathsf{sk}_2}(\mathsf{Decrypt}(\mathsf{sk}_1, c)) = \mathsf{Encrypt}_{\mathsf{sk}_2}(\mathsf{Decrypt}_{\mathsf{sk}_1}(\mathsf{Encrypt}_{\mathsf{sk}_1}(m))) = \mathsf{Encrypt}_{\mathsf{sk}_2}(m)$.

Fig. 2: Bootstrapping: Basic description.

The works that followed Gentry's publication were aimed at proposing new schemes or at improving the bootstrapping in order to make FHE more efficient in practice. The most famous constructions are DGHV [13], BGV [7], GSW [18], and their variants. While the constructions that were successively proposed made the bootstrapping more practical, it still constituted the bottleneck (each bootstrapping taking a few minutes). A much faster bootstrapping, based on a GSW-type scheme, was later devised by Ducas and Micciancio [15],

reducing the bootstrapping time to a sub second. Their technique was further improved and refined, which led to the development of the TFHE scheme [12].

Our techniques and contributions. This paper builds on the state-of-the-art TFHE scheme and extends the TFHE techniques to homomorphically evaluate deep neural networks. TFHE can operate in two modes: leveled and bootstrapped. The *leveled mode* supports linear combinations and a predetermined number of products. The operations evaluated in this mode make the noise always grow. The leveled mode can be used to evaluate small-depth circuits. As for the *bootstrapped mode*, it enables a fine control of the noise by reducing it to a given level whenever it exceeds a certain threshold. Further, as will be shown, the bootstrapped mode is programmable and therefore enables the evaluation of more complex functions. For problems involving circuits of large depth, only the bootstrapped mode is applicable. Deep neural networks belong to that case.

Earlier works attempted to evaluate neural networks using fully homomorphic encryption. Cryptonets [14] was the first initiative towards this goal. They were able to perform a homomorphic inference over 5 layers against the MNIST dataset [24]. In order to limit the noise growth, the standard activation function was replaced with the square function. The underlying encryption scheme was the leveled version of YASHE [4].

A number of subsequent works have adopted a similar approach and improved it in various directions. Among them, it is worth mentioning the results of the iDASH competition [20], whose goal is to find privacy-preserving solutions in the context of genome analysis. The winning solutions of the homomorphic encryption track of the last two editions (namely, [22,3] for 2018 and [21] for 2019) have all in common to rely on *leveled* homomorphic encryption.

Leveled solutions are however inherently limited in the type of tasks they can perform. In particular, in the case of neural networks, they can only accommodate networks with a moderate number of layers. In this paper, we want to emphasize that depth is no longer necessarily an issue and that deep neural networks can actually be evaluated homomorphically. For a specialized type of networks, this was already pointed out in a paper by Bourse *et al.* [6]; specifically, for discretized neural networks whose signals are restricted to the set $\{-1, 1\}$ and where the activation function is the sign function. Central to their scalable construction is an adaptation of the TFHE scheme so as to enable the evaluation of the [non-linear] sign function during a bootstrapping step. In a recent work, Boura *et al.* [5] investigated the applicability of fully homomorphic encryption for *classical* deep neural networks. They simulated the effect of noise propagation by adding a noise value drawn from a normal distribution to intermediate values, while evaluating the model *in the clear*. These experiments were carried out with models making use of the standard ReLU activation function but also with models making use of FHE-friendly variants thereof. Similar experiments were run by replacing max-pooling layers with FHE-friendly average-pooling layers. As a conclusion of their study, the authors of [5] recommend to favor FHE-friendly operations as they appear to be usually more resilient to noise perturbations.

This paper departs from previous works. We do not seek to design new operations or to modify the topology in order to make a given neural network more amenable to an FHE-based implementation. On the contrary, we stick to the original neural network model. This presents the tremendous advantage of not requiring to re-train a new model. As the operations and topology are unchanged, the already trained model can be used as is. The efforts to train a neural network should not be overlooked. This is a costly and time-consuming operation. Furthermore, in many cases, producing a new model is not even possible as this implies having access to the training dataset, which may demand the prior approval of the data owners or of some regulatory authorities.

In our approach, the evaluation of complex functions is achieved thanks to a combination of programmable bootstrapping techniques and leveled operations. Being *programmable* means that any function (including non-linear functions) of an input ciphertext can be obtained as the output of the bootstrapping. Interestingly, the resulting ciphertext features a controlled level of noise. The process can therefore be iterated over and over. So, in the case of machine learning applications, the depth of neural networks can be arbitrarily large. Our techniques are efficient and directly operate on words of a chosen size.

Outline of the paper. The rest of this paper is organized as follows. In the next section, we review the definitions of a number of useful concepts and fix some notation. In Section 3, we introduce a discretized variant of the TFHE scheme. We also detail how to perform basic leveled operations on ciphertexts. Section 4 presents our core technology: the programmable bootstrapping. We proceed in two steps. We start with the regular bootstrapping and, in a second step, generalize it to make it programmable. Section 5 applies the previous results to the homomorphic evaluation of neural networks. We describe a variety of layers that are commonly used to build neural networks and their implementation using our framework. In Section 6, we report numerical experiments and benchmarks. Finally, Section 7 concludes the paper.

2 Preliminaries

2.1 Torus and Torus Polynomials

The letter 'T' in TFHE refers to the real torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$, that is, the set of real numbers modulo 1. Any two elements of $\mathbb{T}$ can be added modulo 1: $(\mathbb{T}, +)$ forms an abelian group. But $\mathbb{T}$ is not a ring as the internal product $\times$ of torus elements is not defined. The *external* product $\cdot$ between integers and torus elements is however well defined. Given $k \in \mathbb{Z}$ and $t \in \mathbb{T}$, the element $k \cdot t \in \mathbb{T}$ is defined as $k \cdot t = t + \cdots + t$ ($|k|$ times) if $k \geq 0$ and $k \cdot t = (-k) \cdot (-t)$ if $k < 0$. Mathematically, $\mathbb{T}$ is endowed with a $\mathbb{Z}$-*module* structure. For any $k, l \in \mathbb{Z}$ and $a, b \in \mathbb{T}$, $(k + l) \cdot a = k \cdot a + l \cdot a$ and $k \cdot (a + b) = k \cdot a + k \cdot b$. Further, the external product is homogeneous: for any $k, l \in \mathbb{Z}$ and $t \in \mathbb{T}$, $k \cdot (l \cdot t) = (kl) \cdot t$.

Polynomials can as well be defined over the torus. As will become apparent, they allow for cryptographic operations otherwise not feasible. Let $\Phi \coloneqq \Phi(X)$ denote the M-th cyclotomic polynomial (i.e., the unique irreducible polynomial with integer coefficients that divides $X^M - 1$ but not $X^k - 1$ for any $k < M$) and let N denote its degree. For performance reasons, M is chosen as a power of 2, in which case it turns out that $N = M/2$ and $\Phi(X) = X^N + 1$. Consider the polynomial rings $\mathbb{R}_N[X] \coloneqq \mathbb{R}[X]/(X^N + 1)$ and $\mathbb{Z}_N[X] \coloneqq \mathbb{Z}[X]/(X^N + 1)$. This defines the $\mathbb{Z}_N[X]$-module

$$\mathbb{T}_N[X] \coloneqq \mathbb{R}_N[X]/\mathbb{Z}_N[X] = \mathbb{T}[X]/(X^N + 1) \ .$$

Elements of $\mathbb{T}_N[X]$ can therefore be seen as polynomials modulo $X^N + 1$ with coefficients in $\mathbb{T}$. Being a $\mathbb{Z}_N[X]$-module, elements in $\mathbb{T}_N[X]$ can be added together and (externally) multiplied by polynomials of $\mathbb{Z}_N[X]$.

Vectors are viewed as row matrices and are denoted with bold letters. Elements in $\mathbb{Z}$ or $\mathbb{T}$ are denoted with roman letters while polynomials are denoted with calligraphic letters. $\mathbb{B}$ is the integer subset $\{0, 1\}$ and, for N a power of 2, $\mathbb{B}_N[X]$ is the subset of polynomials in $\mathbb{Z}_N[X]$ with coefficients in $\mathbb{B}$. For a vector $\boldsymbol{z} \in (\mathbb{Z}/q\mathbb{Z})^n$, its norm $\|\boldsymbol{z}\|$ is defined as the shortest norm among the equivalent classes of $\boldsymbol{z} \in (\mathbb{Z}/q\mathbb{Z})^n$ in $\mathbb{Z}^n$. A polynomial in $\mathbb{Z}_N[X]$ can be identified with a vector in $\mathbb{Z}^N$: to a polynomial $\mathfrak{p} = p_0 + p_1 X + \cdots + p_{N-1} X^{N-1}$ is associated the vector $(p_0, p_1, \ldots, p_{N-1})$. The norm of a polynomial is defined as the norm of its associated vector.

2.2 Probability Distributions

Two probability distributions will be used:

- the uniform distribution, and
- the normal (a.k.a. Gaussian) distribution.

They are respectively denoted by $\mathcal{U}$ and $\mathcal{N}$. When the uniform distribution $\mathcal{U}$ is defined over an interval $[a, b]$, it is written $\mathcal{U}([a, b])$. Discrete intervals are indicated by double brackets; e.g., $[\![a, b]\!]$. The normal distribution is parametrized by its mean μ and its variance σ^2 and is written as $\mathcal{N}(\mu, \sigma^2)$. A normal distribution over the real numbers induces a discretized normal distribution over $\mathbb{Z}/q\mathbb{Z}$: to a real value $X \in \mathbb{R}$ corresponds an integer value $Z = \lceil X\, q \rfloor \pmod q$ with $-q/2 \leq Z \leq q/2$.

If $\mathcal{D}$ is a distribution over a space S then $s \leftarrow \mathcal{D}(S)$ indicates that s is chosen at random in S according to $\mathcal{D}$; $s \xleftarrow{\$} S$ is a shorthand for $s \leftarrow \mathcal{U}(S)$.

2.3 Complexity Assumptions

In 2005, Regev [28] introduced the *learning with errors (LWE) problem*. Generalizations and extensions to ring structures were subsequently proposed in [30,25]. The security of TFHE relies on the hardness of torus-based problems [8,11]: the LWE assumption and the GLWE assumption [7,23] over the torus.

Definition 1 (LWE problem over the torus). *Let $n \in \mathbb{N}$ and let $\boldsymbol{s} = (s_1, \ldots, s_n) \xleftarrow{\$} \mathbb{B}^n$. Let also χ be an error distribution over $\mathbb{R}$. The* learning with errors *(LWE)* over the torus problem *is to distinguish the following distributions:*

$$\mathcal{D}_0 = \left\{(\boldsymbol{a}, r) \mid \boldsymbol{a} \xleftarrow{\$} \mathbb{T}^n, r \xleftarrow{\$} \mathbb{T}\right\}$$

and

$$\mathcal{D}_1 = \left\{(\boldsymbol{a}, r) \mid \boldsymbol{a} = (a_1, \ldots, a_n) \xleftarrow{\$} \mathbb{T}^n, r = \textstyle\sum_{j=1}^n s_j \cdot a_j + e, e \leftarrow \chi\right\} \ .$$

Definition 2 (GLWE problem over the torus). *Let $N, k \in \mathbb{N}$ with N a power of 2 and let $\mathfrak{s} = (\mathfrak{s}_1, \ldots, \mathfrak{s}_k) \xleftarrow{\$} \mathbb{B}_N[X]^k$. Let also χ be an error distribution over $\mathbb{R}_N[X]$. The* general learning with errors *(GLWE)* over the torus problem *is to distinguish the following distributions:*

$$\mathcal{D}_0 = \left\{(\mathfrak{a}, \mathfrak{r}) \mid \mathfrak{a} \xleftarrow{\$} \mathbb{T}_N[X]^k, \mathfrak{r} \xleftarrow{\$} \mathbb{T}_N[X]\right\}$$

and

$$\mathcal{D}_1 = \left\{(\mathfrak{a}, \mathfrak{r}) \mid \mathfrak{a} = (\mathfrak{a}_1, \ldots, \mathfrak{a}_k) \xleftarrow{\$} \mathbb{T}_N[X]^k, \mathfrak{r} = \textstyle\sum_{j=1}^k \mathfrak{s}_j \cdot \mathfrak{a}_j + \mathfrak{e}, \mathfrak{e} \leftarrow \chi\right\} \ .$$

The *decisional LWE assumption* (resp. the *decisional GLWE assumption*) asserts that solving the LWE problem (resp. GLWE problem) is infeasible for some security parameter λ, where $n := n(\lambda)$ and $\chi := \chi(\lambda)$ (resp. $N := N(\lambda)$, $k = k(\lambda)$, and $\chi := \chi(\lambda)$).

3 Discretized TFHE

The LWE assumption over the torus essentially says that a torus element $r \in \mathbb{T}$ constructed as $r = \sum_{j=1}^n s_j \cdot a_j + e$ cannot be distinguished from a *random* torus element $r \in \mathbb{T}$, even if the torus vector $(a_1, \ldots, a_n)$ is given. Torus element $r = \sum_{j=1}^n s_j \cdot a_j + e$ can therefore be used as a random mask to conceal a "plaintext message" $\mu \in \mathbb{T}$ so as to form a ciphertext $\boldsymbol{c} = (a_1, \ldots, a_n, r + \mu) \in \mathbb{T}^{n+1}$, where $s = (s_1, \ldots, s_n) \in \mathbb{B}^n$ plays the role of the private encryption key. The noise e is sampled from a normal error distribution $\chi = \mathcal{N}(0, \sigma^2)$. In the same way, the GLWE assumption gives rise to an encryption mechanism for polynomials of $\mathbb{T}_N[X]$: the encryption of $\mu \in \mathbb{T}_N[X]$ under key $\mathfrak{s} = (\mathfrak{s}_1, \ldots, \mathfrak{s}_k) \in \mathbb{B}_N[X]^k$ being given by $\mathfrak{c} = (\mathfrak{a}_1, \ldots, \mathfrak{a}_k, \mathfrak{r} + \mu) \in \mathbb{T}_N[X]^{k+1}$ with $\mathfrak{r} = \sum_{j=1}^k \mathfrak{s}_j \cdot \mathfrak{a}_j + \mathfrak{e}$. The noise $\mathfrak{e}$ is sampled from a normal error distribution $\chi = \mathcal{N}(0, \sigma^2)$ over $\mathbb{R}_N[X]$; namely, over polynomials of $\mathbb{R}_N[X]$ with coefficients drawn in $\mathcal{N}(0, \sigma^2)$.

As already pointed out in [12], LWE-based ciphertexts can be viewed as a special instance of GLWE-based ciphertexts for $(k, N) = (n, 1)$. Indeed, when $N = 1$, it turns out that $\mathbb{R}_N[X] = \mathbb{R}$, $\mathbb{Z}_N[X] = \mathbb{Z}$, and $\mathbb{T}_N[X] = \mathbb{T}$. Hence, to keep the presentation as general as possible, we will stick to the GLWE setting; LWE-based encryption being a particular case.

In the most generic setting, ciphertexts $\mathfrak{c} = (\mathfrak{a}_1, \ldots, \mathfrak{a}_k, \mathfrak{r} + \mu)$ are vectors of polynomials over $\mathbb{T}_N[X]$. These polynomials can in turn be regarded as vectors over $\mathbb{T}$. In a practical implementation, torus elements are represented with a finite precision (typically, 32 or 64 bits). Let Ω denote the bit-precision—for example, $\Omega = 32$ if the ciphertext components are represented with a precision of 32 bits. In this case, torus elements are restricted to elements of the form $\sum_{i=1}^{\Omega} t_i \cdot 2^{-i} \pmod 1$ with $t_i \in \{0, 1\}$. Essentially, the effect of working with a finite precision boils down to replacing $\mathbb{T}$ with the submodule

$$\hat{\mathbb{T}} := q^{-1}\mathbb{Z}/\mathbb{Z} \subset \mathbb{T} \quad \text{where } q = 2^{\Omega}$$

and doing computations in $\hat{\mathbb{T}}_N[X] \coloneqq \hat{\mathbb{T}}[X]/(X^N+1)$. Viewing $\frac{1}{q}$ as an element in $\hat{\mathbb{T}}_N[X]$, any polynomial $\mu \in \hat{\mathbb{T}}_N[X]$ can be written as

$$\mu = \bar{\mu} \bullet \tfrac{1}{q} \quad \text{for some } \bar{\mu} \in \hat{\mathbb{Z}}_N[X]$$

where $\hat{\mathbb{Z}}_N[X] \coloneqq (\mathbb{Z}/q\mathbb{Z})[X]/(X^N+1)$.

3.1 Encoding/Decoding Messages

The input messages, prior to encryption, can be in any format. The role of the encoding/decoding process is to make them compatible with the encryption scheme.

It is useful to introduce some terminology and notation. An element $\bar{\mu} \in \hat{\mathbb{Z}}_N[X]$ entering the encryption algorithm is referred to as the *plaintext*. It matches a *cleartext* m in a certain finite message space $\mathcal{M}$. The correspondence between cleartexts and plaintexts is given by a message encoding function Encode; the reverse operation is the decoding function Decode. It is required that, for any cleartext $m \in \mathcal{M}$, the relation $\mathsf{Decode}(\mathsf{Encode}(m)) = m$ holds.

Let $\bar{\mu} = \bar{\mu}_0 + \bar{\mu}_1 X + \cdots + \bar{\mu}_{N-1} X^{N-1} \in \hat{\mathbb{Z}}_N[X]$ be a plaintext. Because ciphertexts are noisy and the noise is added to the right (i.e., less significant position), only the upper bits of $\bar{\mu}_i$ are used to encode cleartext messages. In order to successfully complete, certain homomorphic operations demand some of the leading bits of $\bar{\mu}_i$ to be provisioned and set to 0. In the most general case, we let $\varpi \geq 0$ denote the number of these bits (called padding bits) and $\omega \geq 1$ the number of bits that are actually used to represent input cleartexts so that $\varpi + \omega \leq \Omega$. We define $p = 2^{\varpi+\omega}$. Parameter ω is referred to as the cleartext bit-precision and parameter p as the cleartext modulus. The coefficients $\bar{\mu}_i$ of plaintext polynomial $\bar{\mu} \in \hat{\mathbb{Z}}_N[X]$ are therefore of the form $\bar{\mu}_i = 2^{\Omega-(\varpi+\omega)}(\bar{\nu}_i \bmod 2^\omega)$ for some $\bar{\nu}_i$, as shown in Fig. 3.

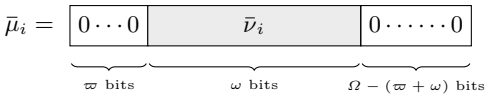


Fig. 3: Plaintext representation.

For an arbitrary element $\bar{x} = \bar{x}_H \frac{q}{p} \pm \bar{x}_L \in \mathbb{Z}/q\mathbb{Z}$ with $0 \leq \bar{x}_L \leq \frac{q}{2p}$, we define the function Upper that returns the value of $\bar{x}_H \frac{q}{p}$. It is specified as $\mathrm{Upper}\colon \mathbb{Z}/q\mathbb{Z} \to \mathbb{Z}/q\mathbb{Z}, \bar{x} \mapsto \mathrm{Upper}(\bar{x})$ with

$$\mathrm{Upper}_{q,p}(\bar{x}) = \frac{q}{p}\left\lceil \frac{p\ \mathrm{lift}(\bar{x})}{q} \right\rfloor \pmod q$$

where the function lift lifts elements of $\mathbb{Z}/q\mathbb{Z}$ to $\mathbb{Z}$. The function Upper naturally extends to polynomials of $\hat{\mathbb{Z}}_N[X]$ by applying it coefficient-wise.

In particular, for $q = 2^\Omega$ and $p = 2^{\varpi+\omega}$, the function reads as $\mathrm{Upper}_{q,p}(\bar{x}) = 2^{\Omega-(\varpi+\omega)}\left\lceil \frac{2^{\varpi+\omega}\ \mathrm{lift}(\bar{x})}{2^\Omega} \right\rfloor \pmod q$. Note that if $\bar{x} = \bar{x}_H\, 2^{\Omega-(\varpi+\omega)} \pm \bar{x}_L \in \mathbb{Z}/q\mathbb{Z}$ with $0 \leq \bar{x}_L \leq 2^{\Omega-(\varpi+\omega)-1}$ then $\mathrm{Upper}_{q,p}(\bar{x}) = \bar{x}_H\, 2^{\Omega-(\varpi+\omega)}$.

3.2 Description

We are now ready to present the implementation of TFHE encryption with a finite representation precision. We write $\overline{\mathrm{GLWE}}$ the corresponding encryption algorithm in the general case. The $\overline{\mathrm{LWE}}$ encryption algorithm coincides with the particular case $(k, N) = (n, 1)$.

$\mathsf{KeyGen}(1^\lambda)$ On input security parameter λ, define a pair of integers (k, N) with $k \geq 1$ and N a power of 2. Define also a normal error distribution $\chi = \mathcal{N}(0, \sigma^2)$ over $\mathbb{R}_N[X]$. Sample uniformly at random a vector $\mathfrak{z} = (\mathfrak{z}_1, \ldots, \mathfrak{z}_k) \xleftarrow{\$} \mathbb{B}_N[X]^k$. The plaintext space is $\hat{\mathbb{Z}}_N[X] = (\mathbb{Z}/q\mathbb{Z})[X]/(X^N + 1)$ where $q = 2^\Omega$ and the cleartext modulus is $p = 2^{\varpi+\omega}$, with $\Omega \geq \varpi + \omega$.

The public parameters are $\mathsf{pp} = \{k, N, \sigma, p, q\}$ and the private key is $\mathsf{sk} = \mathfrak{z}$.

$\mathsf{Encrypt}_{\mathsf{sk}}(\bar{\mu})$ The encryption of a plaintext $\bar{\mu} \in \hat{\mathbb{Z}}_N[X]$ is given by

$$\bar{\mathbf{c}} \leftarrow \overline{\mathrm{GLWE}}_{\mathfrak{z}}(\bar{\mu}) \coloneqq (\bar{\alpha}_1, \ldots, \bar{\alpha}_k, \bar{b}) \in \hat{\mathbb{Z}}_N[X]^{k+1}$$

with

$$\begin{cases} \bar{\mu}^* = \bar{\mu} + \bar{e} \pmod{(q, X^N + 1)} \\ \bar{b} = \sum_{j=1}^k \mathfrak{z}_j \, \bar{\alpha}_j + \bar{\mu}^* \pmod{(q, X^N + 1)} \end{cases}$$

for a random polynomial vector $(\bar{\alpha}_1, \ldots, \bar{\alpha}_k) \xleftarrow{\$} \hat{\mathbb{Z}}_N[X]^k$ and a discrete noise $\bar{e} = \lceil e\, q \rfloor \pmod q$ for some $e \leftarrow \mathbb{R}_N[X]$ whose coefficients are sampled in $\mathcal{N}(0, \sigma^2)$.

$\mathsf{Decrypt}_{\mathsf{sk}}(\bar{\mathbf{c}})$ To decrypt $\bar{\mathbf{c}} = (\bar{\alpha}_1, \ldots, \bar{\alpha}_k, \bar{b})$, use private key $\mathfrak{z} = (\mathfrak{z}_1, \ldots, \mathfrak{z}_k)$, compute in $\hat{\mathbb{Z}}_N[X]$

$$\bar{\mu}^* = \bar{b} - \sum_{j=1}^k \mathfrak{z}_j \, \bar{\alpha}_j \pmod{(q, X^N + 1)} ,$$

and output $\mathrm{Upper}_{q,p}(\bar{\mu}^*)$.

Correctness. Let $p = 2^{\varpi+\omega}$, $q = 2^\Omega$ and $\bar{\mu} = \bar{\mu}_0 + \cdots + \bar{\mu}_{N-1} X^{N-1}$ with $\bar{\mu}_i = \frac{q}{p} (\bar{\nu}_i \bmod 2^\omega)$ for some $\bar{\nu}_i$; see Section 3.1. It can be verified that if $\bar{\mathbf{c}} \leftarrow \mathsf{Encrypt}_{\mathfrak{z}}(\bar{\mu})$ then $\mathsf{Decrypt}_{\mathfrak{z}}(\bar{\mathbf{c}}) = \bar{\mu}$, provided that $\|\bar{e}\|_\infty < \frac{q}{2p} = 2^{\Omega-(-\varpi+\omega)-1}$.

Proof. By construction, we have $\bar{\mu}^* = \bar{\mu} + \bar{e}$ where $\bar{e} = \bar{e}_0 + \cdots + \bar{e}_{N-1} X^{N-1}$. Hence, lifting $\bar{\mu}^*$ to $\mathbb{Z}_N[X]$, we have $\left\lceil \frac{p}{q} \bar{\mu}^* \right\rfloor = \left\lceil \frac{p}{q} (\bar{\mu} + \bar{e}) \right\rfloor = \sum_{i=0}^{N-1} \left\lceil \frac{p}{q} (\bar{\mu}_i + \bar{e}_i \pmod q) \right\rfloor X^i = \sum_{i=0}^{N-1} \left\lceil \frac{p}{q} (\bar{\mu}_i + \bar{e}_i + q\, \delta_i) \right\rfloor X^i = \sum_{i=0}^{N-1} \big((\bar{\nu}_i \bmod 2^\omega) + \left\lceil \frac{p}{q} \bar{e}_i \right\rfloor + p\, \delta_i \big) X^i$ for some $\delta_i \in \mathbb{Z}$. Noting that $\left\lceil \frac{p}{q} \bar{e}_i \right\rfloor = 0$ if $\|\bar{e}\|_\infty < \frac{q}{2p}$, it follows that $\mathrm{Upper}_{q,p}(\bar{\mu}^*) \equiv \frac{q}{p} \left\lceil \frac{p}{q} \bar{\mu}^* \right\rfloor \equiv \frac{q}{p} \sum_{i=0}^{N-1} (\bar{\nu}_i \bmod 2^\omega) X^i \equiv \bar{\mu} \pmod q$.

Remark 1. In certain applications, it is acceptable that the decryption algorithm does not recover the *exact* initial plaintext but a close approximation thereof. In this case, the requirement becomes $\mathsf{Decrypt}_{\mathfrak{z}}(\bar{\mathbf{c}}) \approx \bar{\mu}$ and the condition on the bound of $\|\bar{e}\|_\infty$ can be relaxed.

3.3 Leveled Operations

FHE enables directly performing operations on ciphertexts. Depending on the type of operation, the resulting noise level increases more or less.

Addition. Clearly, $\overline{\mathrm{GLWE}}$ ciphertexts are homomorphic with respect to the addition. Let $\bar{\mathbf{c}}_1 \leftarrow \overline{\mathrm{GLWE}}_{\mathfrak{z}}(\bar{\mu}_1)$ and $\bar{\mathbf{c}}_2 \leftarrow \overline{\mathrm{GLWE}}_{\mathfrak{z}}(\bar{\mu}_2)$ with $\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2 \in \hat{\mathbb{Z}}_N[X]^{k+1}$ be the respective encryptions of plaintexts $\bar{\mu}_1, \bar{\mu}_2 \in \hat{\mathbb{Z}}_N[X]$; i.e.,

$$\bar{\mathbf{c}}_i = \left(\bar{\alpha}_1^{(i)}, \ldots, \bar{\alpha}_k^{(i)}, \bar{b}_i\right) \qquad (i \in \{1, 2\})$$

with $\bar{b}_i = \sum_{j=1}^k \mathfrak{z}_j \, \bar{\alpha}_j^{(i)} + \bar{\mu}_i + \bar{e}_i$. Then

$$\begin{aligned} \bar{\mathbf{c}}_3 &= \bar{\mathbf{c}}_1 + \bar{\mathbf{c}}_2 \\ &= \left(\bar{\alpha}_1^{(1)} + \bar{\alpha}_1^{(2)}, \ldots, \bar{\alpha}_k^{(1)} + \bar{\alpha}_k^{(2)}, \bar{b}_1 + \bar{b}_2\right) \end{aligned}$$

is a $\overline{\mathrm{GLWE}}$ encryption of $(\bar{\mu}_1 + \bar{\mu}_2) \in \hat{\mathbb{Z}}_N[X]$, provided that the resulting noise $\bar{e}_3 = \bar{e}_1 + \bar{e}_2$ keeps small.

Scalar multiplication. By extension, $\overline{\mathrm{GLWE}}$ ciphertexts are homomorphic with respect to the multiplication by a constant. Let $K \in \mathbb{Z}_{\geq 0}$ and $\bar{\mathbf{c}} \leftarrow \overline{\mathrm{GLWE}}_{\mathfrak{z}}(\bar{\mu}) = (\bar{a}_1, \ldots, \bar{a}_k, \bar{b})$ with $\bar{b} = \sum_{j=1}^{k} \mathfrak{z}_j \, \bar{a}_j + \bar{\mu} + \bar{e}$. Then

$$K \cdot \bar{\mathbf{c}} = \underbrace{\bar{\mathbf{c}} + \cdots + \bar{\mathbf{c}}}_{K \text{ times}}$$

is an encryption of $K \cdot \bar{\mu} \in \hat{\mathbb{Z}}_N[X]$, provided that $K \cdot \bar{e}$ keeps small. If $K < 0$ then $K \cdot \bar{\mathbf{c}} = (-K) \cdot (-\bar{\mathbf{c}})$. More generally, if $\mathcal{K} \in \mathbb{Z}_N[X]$ then $\mathcal{K} \cdot \bar{\mathbf{c}}$ is an encryption of $\mathcal{K} \cdot \bar{\mu} \in \hat{\mathbb{Z}}_N[X]$, provided that $\mathcal{K} \cdot \bar{e}$ keeps small.

External product. $\overline{\mathrm{GLWE}}$ ciphertexts do not support a native internal multiplication which, in practice, means that two $\overline{\mathrm{GLWE}}$ ciphertexts cannot be directly multiplied. In order to perform a multiplication, a clever matrix-based approach put forward in the GSW construction [18] can be used. By analogy, we write $\overline{\mathrm{GGSW}}$ the corresponding encryption algorithm in the general case; the particular case $(k, N) = (n, 1)$ is denoted by $\overline{\mathrm{GSW}}$.

With the previous $\overline{\mathrm{GLWE}}$ notation, let parameters $B = 2^\beta$ and ℓ with $\beta, \ell \geq 1$ and such that $\ell\beta \leq \Omega$. Define also the vector $\boldsymbol{g} = (2^{\Omega-\beta}, 2^{\Omega-2\beta}, \ldots, 2^{\Omega-\ell\beta})$. The $\overline{\mathrm{GGSW}}$ encryption of a plaintext $m \in \mathbb{Z}_N[X]$ with respect to a $\overline{\mathrm{GLWE}}$ encryption key $\mathfrak{z} \in \mathbb{B}_N[X]^k$ is defined as

$$\begin{aligned} \overline{\mathbf{C}} &\leftarrow \overline{\mathrm{GGSW}}_{\mathfrak{z}}(m) \\ &\coloneqq \overline{\mathbf{Z}} + m \cdot \mathbf{G}^\intercal \in \hat{\mathbb{Z}}_N[X]^{(k+1)\ell \times (k+1)} \end{aligned}$$

where

$$\overline{\mathbf{Z}} \leftarrow \begin{pmatrix} \overline{\mathrm{GLWE}}_{\mathfrak{z}}(0) \\ \overline{\mathrm{GLWE}}_{\mathfrak{z}}(0) \\ \vdots \\ \overline{\mathrm{GLWE}}_{\mathfrak{z}}(0) \end{pmatrix} \Bigg\} (k+1)\ell \text{ rows}$$

is a matrix containing on each row a fresh $\overline{\mathrm{GLWE}}$ encryption of 0 and where

$$\begin{aligned} \mathbf{G}^\intercal &= \mathbf{I}_{k+1} \otimes \boldsymbol{g}^\intercal \\ &= \mathrm{diag}(\underbrace{\boldsymbol{g}^\intercal, \ldots, \boldsymbol{g}^\intercal}_{k+1}) \in \mathbb{Z}_N[X]^{(k+1)\ell \times (k+1)} \end{aligned}$$

is the so-called *gadget matrix* [26], with $\mathbf{I}_{k+1}$ the identity matrix of size $k+1$. It is worth noting that any element d in $\mathbb{Z}/q\mathbb{Z}$, viewed as an integer in $[\![-\frac{q}{2}, \frac{q}{2}]\!]$, can always be approximated by a signed-digit radix-B expansion of size ℓ as

$$d \approx q \sum_{i=1}^{\ell} d_i \, B^{-i} = \sum_{i=1}^{\ell} d_i \, 2^{\Omega - i\beta} = \boldsymbol{g}^{-1}(d) \, \boldsymbol{g}^\intercal$$

where $\boldsymbol{g}^{-1}(d) \coloneqq (d_1, \ldots, d_\ell) \in \mathbb{Z}^\ell$ with digits $d_i \in [\![-\frac{B}{2}, \frac{B}{2}]\!]$ and with an approximation error that is bounded by $\left| \boldsymbol{g}^{-1}(d) \, \boldsymbol{g}^\intercal - d \right| \leq q/(2B^\ell) = 2^{\Omega - \beta\ell - 1}$.

By extension, for a polynomial $\mathfrak{p} = p_0 + \cdots + p_{N-1} \, X^{N-1} \in \hat{\mathbb{Z}}_N[X]$ whose coefficients are viewed as integers in $[\![-\frac{q}{2}, \frac{q}{2}]\!]$, the decomposition $\boldsymbol{g}^{-1}(\mathfrak{p}) \in \mathbb{Z}_N[X]^\ell$ is defined as $\boldsymbol{g}^{-1}(\mathfrak{p}) = \sum_{j=0}^{N-1} \boldsymbol{g}^{-1}(p_j) \, X^j$. Clearly, it holds that $\left\| \boldsymbol{g}^{-1}(\mathfrak{p}) \, \boldsymbol{g}^\intercal - \mathfrak{p} \right\|_\infty \leq 2^{\Omega - \beta\ell - 1}$. Finally, for a vector of $k+1$ polynomials, $\boldsymbol{\mathfrak{p}} = (\mathfrak{p}_1, \ldots, \mathfrak{p}_{k+1}) \in \hat{\mathbb{Z}}_N[X]^{k+1}$, the decomposition $\mathbf{G}^{-1}(\boldsymbol{\mathfrak{p}}) \in \mathbb{Z}_N[X]^{(k+1)\ell}$ is defined as $\mathbf{G}^{-1}(\boldsymbol{\mathfrak{p}}) = (\boldsymbol{g}^{-1}(\mathfrak{p}_1), \ldots, \boldsymbol{g}^{-1}(\mathfrak{p}_{k+1}))$ and $\left\| \mathbf{G}^{-1}(\boldsymbol{\mathfrak{p}}) \, \mathbf{G}^\intercal - \boldsymbol{\mathfrak{p}} \right\|_\infty \leq 2^{\Omega - \beta\ell - 1}$.

Interestingly, the gadget decomposition of $\overline{\mathrm{GLWE}}$ ciphertexts gives rise to an *external product* [12] with $\overline{\mathrm{GGSW}}$ ciphertexts. Specifically, for plaintexts $m_1 \in \mathbb{Z}_N[X]$ and $\mu_2 \in \hat{\mathbb{Z}}_N[X]$, if $\overline{\mathbf{C}}_{\mathbf{1}} \leftarrow \overline{\mathrm{GGSW}}_{\mathfrak{z}}(m_1)$ and $\bar{\mathbf{c}}_{\mathbf{2}} \leftarrow \overline{\mathrm{GLWE}}(\mu_2)$ then their external product, denoted by $\boxdot$, is given by

$$\bar{\mathbf{c}}_{\mathbf{3}} = \overline{\mathbf{C}}_{\mathbf{1}} \boxdot \bar{\mathbf{c}}_{\mathbf{2}} \coloneqq \mathbf{G}^{-1}(\bar{\mathbf{c}}_{\mathbf{2}}) \, \overline{\mathbf{C}}_{\mathbf{1}} \quad .$$

A little algebra shows that

$$\begin{aligned}\bar{\mathbf{c}}_3 &= \mathbf{G}^{-1}(\bar{\mathbf{c}}_2)\,(\overline{\mathfrak{E}} + m_1 \cdot \mathbf{G}^{\intercal}) \\ &= \underbrace{\mathbf{G}^{-1}(\bar{\mathbf{c}}_2)\,\overline{\mathfrak{E}}}_{=\overline{\mathrm{GLWE}}_{\mathfrak{s}}(0)} + \underbrace{m_1 \cdot \mathbf{G}^{-1}(\bar{\mathbf{c}}_2)\,\mathbf{G}^{\intercal}}_{\approx m_1\,\bar{\mathbf{c}}_2}\end{aligned}$$

is a $\overline{\mathrm{GLWE}}$ encryption of $m_1\,\mu_2 \in \hat{\mathbb{Z}}_N[X]$, provided that the resulting noise (including the approximation error) keeps small.

The CMux gate. Starting from the external product, a new leveled operation can be defined: the 'controlled' multiplexer or CMux [12, § 3.4]. A CMux acts as a selector according to a bit—but over encrypted data. It takes as input two $\overline{\mathrm{GLWE}}$ ciphertexts $\bar{\mathbf{c}}_0$ and $\bar{\mathbf{c}}_1$, respectively encrypting plaintexts μ_0 and $\mu_1 \in \hat{\mathbb{Z}}_N[X]$, and a $\overline{\mathrm{GGSW}}$ ciphertext $\overline{\mathscr{C}}$ encrypting a bit b. The result is a $\overline{\mathrm{GLWE}}$ ciphertext $\bar{\mathbf{c}}'$ encrypting μ_b, provided that the resulting noise keeps small. The CMux gate is given by:

$$\bar{\mathbf{c}}' \leftarrow \mathsf{CMux}(\overline{\mathscr{C}}, \bar{\mathbf{c}}_0, \bar{\mathbf{c}}_1) \coloneqq \overline{\mathscr{C}} \boxdot (\bar{\mathbf{c}}_1 - \bar{\mathbf{c}}_0) + \bar{\mathbf{c}}_0 \ .$$

It plays a central role in the performance of homomorphic computations and, especially, inside the bootstrapping.

4 Programmable Bootstrapping

The *programmable bootstrapping* is an extension of the bootstrapping technique that allows resetting the noise to a fixed level while—at the same time—evaluating a function on the input ciphertext.

In this section, we first explain in detail how to perform the regular bootstrapping. We then proceed with the programmable bootstrapping and show how to evaluate any function expressed as a look-up table. When f is the identity function, that coincides with a regular bootstrapping.

4.1 Blind Rotation

As aforementioned, Gentry's bootstrapping boils down to homomorphically decrypt a ciphertext using a homomorphic encryption of its own decryption key, with the goal of reducing the noise the ciphertext contains.

Intuition. Consider an $\overline{\mathrm{LWE}}$ ciphertext $\bar{\boldsymbol{c}} \leftarrow \overline{\mathrm{LWE}}_{\boldsymbol{s}}(\bar{\mu}) = (\bar{a}_1, \ldots, \bar{a}_n, \bar{b}) \in (\mathbb{Z}/q\mathbb{Z})^{n+1}$ where $\bar{a}_j \xleftarrow{\$} \mathbb{Z}/q\mathbb{Z}$ and $\bar{b} = \sum_{j=1}^{n} s_j\,\bar{a}_j + \bar{\mu}^*$ with $\bar{\mu}^* = \bar{\mu} + \bar{e}$ for some discrete noise $\bar{e} = \lceil e\,q \rfloor \pmod q$ with $e \leftarrow \mathcal{N}(0, \sigma^2)$. Ciphertext $\bar{\boldsymbol{c}}$ is an encryption of plaintext $\bar{\mu} \in \mathbb{Z}/q\mathbb{Z}$ under the secret key $\boldsymbol{s} = (s_1, \ldots, s_n) \in \mathbb{B}^n$. It can be decrypted using $\boldsymbol{s}$ in two steps as $\bar{\mu}^* \leftarrow \bar{b} - \sum_{j=1}^{n} s_j\,\bar{a}_j$ and $\bar{\mu} \leftarrow \mathrm{Upper}_{q,p}(\bar{\mu}^*)$.

In order to bootstrap, one way to look at the decryption (without the rounding) is to see that

$$-\bar{\mu}^* = -\bar{b} + \textstyle\sum_{j=1}^{n} s_j\,\bar{a}_j \pmod q$$

and to put this value at the exponent of X to get the monomial $X^{-\bar{\mu}^*}$. Note that there are q possible values for $\bar{\mu}^*$. The rough idea (more technical details are given later) is then to build a polynomial—that we call *test polynomial*—such that each one of its coefficients encodes the noise-free value corresponding to $\bar{\mu}^*$ (namely, $\bar{\mu} = \mathrm{Upper}_{q,p}(\bar{\mu}^*)$), for all the possible $\bar{\mu}^*$'s. Specifically, if we suppose for a moment that the test polynomial is the degree-q polynomial $\bar{\mathfrak{v}} = \bar{v}_0 + \bar{v}_1\,X + \cdots + \bar{v}_{q-1}\,X^{q-1}$ then its i^{th} coefficient is set to $\bar{v}_i = \mathrm{Upper}_{q,p}(i \bmod q)$. By rotating the test polynomial of $\bar{\mu}^*$ positions, the value of $\bar{\mu}$ moves to the constant coefficient position. This is illustrated in Fig. 4. It then remains to extract it.

Of course, this rotation is done homomorphically (hence the name blind rotation) and since $X^{-\bar{\mu}^*} \cdot \bar{\mathfrak{v}}$ is a polynomial, this is where $\overline{\mathrm{GLWE}}$ encryption comes into play.

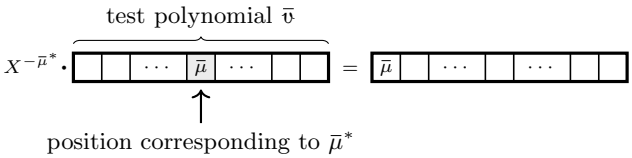


Fig. 4: Rotating the test polynomial.

Polynomials used in $\overline{\text{GLWE}}$ are defined modulo X^N+1. This means that X as a multiplicative element of $\mathbb{Z}_N[X]$ is of order $2N$. However, as appearing in the $\overline{\text{LWE}}$ encryption, $\bar{\mu}^*$ is defined modulo q. It therefore needs to be rescaled modulo $2N$. As a consequence, instead of using the relation $-\bar{\mu}^* = -\bar{b} + \sum_{j=1}^{n} s_j\,\bar{a}_j \pmod q$, we have to rely on the approximation

$$-\tilde{\mu}^* = -\tilde{b} + \textstyle\sum_{j=1}^{n} s_j\,\tilde{a}_j \pmod{2N}\,,$$

where $\tilde{b} = \left\lceil \frac{2N(\bar{b} \bmod q)}{q} \right\rfloor$ and $\tilde{a}_j = \left\lceil \frac{2N(\bar{a}_j \bmod q)}{q} \right\rfloor$. This approximation may generate a small additional error that adds to the noise. We call this additional error *drift*. It depends on both the size n of the input $\overline{\text{LWE}}$ and the ring size N used in $\overline{\text{GLWE}}$ during the rotation. The impact of the drift on the result can be dealt with by a careful choice of the parameters.

Also, because the test polynomial $\bar{\mathfrak{v}}$ lies in $\hat{\mathbb{Z}}_N[X]$ and thus has N coefficients, at most N values for $\tilde{\mu}^*$ can be encoded. This is addressed by ensuring that the most significant bit of $\tilde{\mu}^*$ is set to 0; that is, parameter $\varpi \geq 1$ (see Section 3.1). In this case, $\tilde{\mu}^*$ can take at most N possible values and the test polynomial is formed as $\bar{\mathfrak{v}} = \bar{v}_0 + \cdots + \bar{v}_{N-1}\,X^{N-1}$ with

$$\bar{v}_i = \text{Upper}_{q,p}\big(\tfrac{q}{2N}\,i \bmod q\big)\ .$$

Implementation. It remains to explain how to compute the product between $X^{-\tilde{\mu}^*}$ and the test polynomial $\bar{\mathfrak{v}}$ under $\overline{\text{GLWE}}$ encryption. It turns out that such a computation can be computed as a succession of CMux gates [12, § 4.3].

We use the previous notations. The secret key bits s_j used to encrypt the input $\overline{\text{LWE}}$ ciphertext cannot be revealed. They are instead provided as *bootstrapping keys*; i.e., encrypted under some encryption key $\mathfrak{z}' \in \mathbb{B}_N[N]^k$:

$$\mathsf{bsk}[j] \leftarrow \overline{\text{GGSW}}_{\mathfrak{z}'}(s_j) \quad \text{for all } j = 1,\ldots,n\ .$$

We then have:

Algorithm 1: Blind rotation.

ACC $\leftarrow (0,\ldots,0,X^{-\tilde{b}}\cdot\bar{\mathfrak{v}})$
for $i=1$ **to** n **do**
  ACC $\leftarrow \mathsf{CMux}(\mathsf{bsk}[i],\text{ACC},X^{\tilde{a}_i}\cdot\text{ACC})$
end for
return ACC

At the end of the loop, ACC contains a $\overline{\text{GLWE}}$ encryption of $X^{-\tilde{\mu}^*}\cdot\bar{\mathfrak{v}}$ under key $\mathfrak{z}'$.

Indeed, let $\bar{\boldsymbol{c}}'_j$ denote the value of the accumulator ACC at iteration $i = j$ in Algorithm 1. It turns out that $\bar{\boldsymbol{c}}'_j \leftarrow \overline{\mathrm{GLWE}}_{\mathfrak{z}'}(X^{-\tilde{b}+\sum_{i=1}^{j} s_i\,\tilde{a}_i} \cdot \bar{\mathfrak{v}})$. As a consequence, letting $\bar{\boldsymbol{c}}' = \bar{\boldsymbol{c}}'_n$, we then get

$$\begin{aligned}\bar{\boldsymbol{c}}' &\leftarrow \underbrace{\overline{\mathrm{GLWE}}_{\mathfrak{z}'}(X^{-\tilde{b}+\sum_{i=1}^{n} s_i\,\tilde{a}_i} \cdot \bar{\mathfrak{v}})}_{=\overline{\mathrm{GLWE}}_{\mathfrak{z}'}(X^{-\tilde{\mu}^*}\cdot\bar{\mathfrak{v}})} \\ &\leftarrow \overline{\mathrm{GLWE}}_{\mathfrak{z}'}(\bar{\mathfrak{u}})\end{aligned}$$

for some polynomial $\bar{\mathfrak{u}} \in \hat{\mathbb{Z}}_N[X]$ whose constant term is $\bar{\mu}$; i.e., $\bar{\mathfrak{u}} \coloneqq X^{-\tilde{\mu}^*} \cdot \bar{\mathfrak{v}} = \bar{u}_0 + \bar{u}_1\,X + \cdots + \bar{u}_{N-1}\,X^{N-1}$ where $\bar{u}_0 = \bar{\mu}$, as desired.

The remaining step of the bootstrapping consists in extracting the constant coefficient of $\bar{\mathfrak{u}} = X^{-\tilde{\mu}^*} \cdot \bar{\mathfrak{v}}$ as a $\overline{\mathrm{LWE}}$ ciphertext of $\bar{\mu}$. This is an easy operation—called *sample extraction*—which is performed by simply extracting some of the coefficients of the $\overline{\mathrm{GLWE}}$ ciphertext.

In more detail, let $\boldsymbol{\mathfrak{z}}' = (\mathfrak{z}'_1, \ldots, \mathfrak{z}'_k) \in \mathbb{B}_N[X]^k$ with $\mathfrak{z}'_j = s'_{j,0} + \cdots + s'_{j,N-1}\,X^{N-1}$ for $1 \le j \le k$. Parsing $\bar{\boldsymbol{c}}' \leftarrow \overline{\mathrm{GLWE}}_{\mathfrak{z}'}(\bar{\mathfrak{u}}) \in \hat{\mathbb{Z}}_N[X]^k$ as $(\bar{a}'_1, \ldots, \bar{a}'_k, \bar{\mathfrak{b}}')$ with $\bar{a}'_j = \bar{a}'_{j,0} + \cdots + \bar{a}'_{j,N-1}\,X^{N-1}$ for $1 \le j \le k$ and $\bar{\mathfrak{b}}' = \bar{b}'_0 + \cdots + \bar{b}'_{N-1}\,X^{N-1}$, it can be verified that $\bar{\boldsymbol{c}}' \coloneqq (\bar{a}'_{1,0}, -\bar{a}'_{1,N-1}, \ldots, -\bar{a}'_{1,1}, \ldots, \bar{a}'_{k,0}, -\bar{a}'_{k,N-1}, \ldots, -\bar{a}'_{k,1}, \bar{b}'_0) \in (\mathbb{Z}/q\mathbb{Z})^{kN+1}$ is a $\overline{\mathrm{LWE}}$ encryption of $\bar{\mu}$ under the key $\boldsymbol{s}' = (s'_1, \ldots, s'_{kN}) \in \mathbb{B}^{kN}$ where $s'_{l+1+(j-1)N} \coloneqq s'_{j,l}$ for $1 \le j \le k$ and $0 \le l \le N-1$.

Key switching. The loop is almost closed. With the above procedure, input ciphertext $\bar{\boldsymbol{c}} \leftarrow \overline{\mathrm{LWE}}_{\boldsymbol{s}}(\bar{\mu}) \in (\mathbb{Z}/q\mathbb{Z})^{n+1}$ and resulting output ciphertext $\bar{\boldsymbol{c}}' \leftarrow \overline{\mathrm{LWE}}_{\boldsymbol{s}'}(\bar{\mu}) \in (\mathbb{Z}/q\mathbb{Z})^{kN+1}$ both encrypt plaintext $\bar{\mu}$ but make use of different keys and have a different format.

In order to convert $\bar{\boldsymbol{c}}'$ back to the original setting, an operation called *key switching* can be performed. The key-switching technique is classical in FHE; it can be used to switch encryption keys in different parameter sets [9, § 1.2]. Its implementation requires *key-switching keys*: they essentially consist in an $\overline{\mathrm{LWE}}$ encryption of the key bits of $\boldsymbol{s}'$ with respect to the original key $\boldsymbol{s}$. The procedure may seem conceptually very similar to the bootstrapping, but there is a fundamental difference between the two techniques: bootstrapping reduces the noise (and is computationally demanding) whereas the key switching makes the noise increase (but is cheaper to evaluate). See Appendix A for details.

Remark 2. As described above, the input ciphertext undergoes a series of three transformations—blind rotation, sample extraction, key switching—in that order; or schematically,

$$\bar{\boldsymbol{c}}^{(\mathrm{out})} \leftarrow \mathsf{KeySwitch} \circ \mathsf{SampleExtract} \circ \mathsf{BlindRotate}(\bar{\boldsymbol{c}}^{(\mathrm{in})}) \ .$$

In [6], the authors astutely notice that the [noisy] key switching procedure can be moved around. If we instead consider input/output ciphertexts in $(\mathbb{Z}/q\mathbb{Z})^{kN+1}$ then the bootstrapping process appears as the succession $\bar{\boldsymbol{c}}'^{(\mathrm{out})} \leftarrow \mathsf{SampleExtract} \circ \mathsf{BlindRotate} \circ \mathsf{KeySwitch}(\bar{\boldsymbol{c}}'^{(\mathrm{in})})$. Since the sample extraction is noiseless, the output ciphertext contains uniquely the noise resulting from the blind rotation. The noise corresponding to the key switching does not add up.

4.2 Look-up Table Evaluation

In the previous section, the blind rotation is used to perform a bootstrapping. Surprisingly, the same technique can be adapted so as to evaluate a function at the same time. The function is evaluated as a look-up table that is encoded in the test polynomial.

Specifically, suppose we intend to evaluate—over encrypted data—an arbitrary function f with domain $\mathfrak{D}$ and image $\mathfrak{I}$, $f\colon \mathfrak{D} \to \mathfrak{I}, x \mapsto y = f(x)$. We assume we are given the encoding functions $\mathsf{Encode}\colon \mathfrak{D} \to \mathbb{Z}/q\mathbb{Z}$ and $\mathsf{Encode}'\colon \mathfrak{I} \to \mathbb{Z}/q\mathbb{Z}$, and the matching decoding functions Decode and Decode', as specified in Section 3.1.

We showed in the previous section that selecting for the test polynomial $\bar{\mathfrak{v}} = \bar{v}_0 + \cdots + \bar{v}_{N-1}\,X^{N-1}$ with $\bar{v}_i = \mathrm{Upper}_{q,p}\big(\frac{q}{2N}\,i \bmod q\big)$ transforms a ciphertext of $\bar{\mu}$ into another ciphertext of $\bar{\mu}$ with a lesser noise.

Consider now the following diagram

$$\begin{array}{ccc} \mathbb{Z}/q\mathbb{Z} & \longrightarrow & \mathbb{Z}/q\mathbb{Z} \\ \mathsf{Decode}\downarrow & & \uparrow\mathsf{Encode}' \\ \mathfrak{D} & \xrightarrow{\quad f \quad} & \mathfrak{F} \end{array}$$

and suppose we define a look-up table as pairs $(i, T[i])$ for $0 \leq i \leq N-1$ with

$$T[i] = \mathsf{Encode}' \circ f \circ \mathsf{Decode} \circ \mathrm{Upper}_{q,p}\left(\tfrac{q}{2N}\, i \bmod q\right) \ .$$

As the diagram suggests, we program the test polynomial as

$$\bar{\mathfrak{v}} = \bar{v}_0 + \cdots + \bar{v}_{N-1}\, X^{N-1} \quad \text{with } \bar{v}_i = T[i]\,,$$

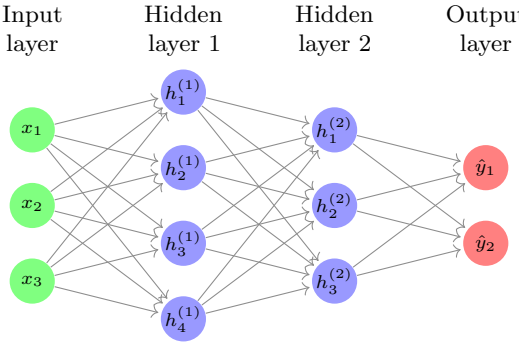
the rest of the process described in Section 4.1 remains unchanged. Doing so, up to the drift, an input ciphertext of $\bar{\mu}$ (encoding some value $x \in \mathfrak{D}$) will be transformed into a ciphertext of a value encoding $f(x)$. Furthermore, being the output of a bootstrapping, the resulting ciphertext enjoys a low level of noise. The whole process is what we call *programmable bootstrapping*.

Remark 3. As explained, the regular bootstrapping requires an encoding parameter $\varpi \geq 1$ (cf. Section 4.1). This condition can be lifted in the programmable bootstrapping when the entries of the look-up table are negacyclic; i.e., when $T[i+N] \equiv -T[i] \pmod{2N}$. In that case, $2N$ values are actually programmed.

5 Application to Neural Networks

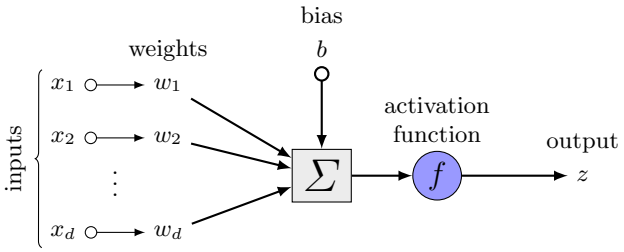
All the tools that we need are now on the table. In this section, we apply them in order to evaluate homomorphically neural networks.

Neural networks (NN) were originally built in computer science by analogy to the human brain in order to solve complex problems that machines were not able to solve before. The neural networks can be trained and then used to classify objects, detect diseases, do face recognition, and so on. An example of a neural network is presented in Fig. 5. Figure 6 depicts an artificial neuron.



A neural network is composed of interconnected artificial neurons, and these neurons are organized into layers. The first layer contains the initial data and is called input layer and the final layer is called output layer. The layers between them is the place where all the computation is done; they are are known as hidden layers.

Fig. 5: A simple neural network with 2 hidden layers.



Artificial neurons mimic their biological counterparts. A neuron takes on input d signals x_i, each signal being weighted by a parameter w_i (quantifying their respective impact), and a bias parameter b. The output signal is $z = f(s)$ where $s = \sum_{i=1}^{d} w_i\, x_i + b$. Function f is non-linear and is referred to as activation function.

Fig. 6: Artificial neuron.

The different layers in a neural network are typically aimed at successively extracting discriminating features or patterns from the input data. The number of layers and the type of operations that is performed in each layer depend on the task the neural network is trying to achieve.

We review below a number of layers that are commonly used to build neural networks. The list is non-exhaustive. Our techniques are generic and support all known types of layers. Each layer receives inputs from the previous layer, performs some computations, and produces outputs. The outputs then flow to the next layer as inputs. Two types of layers are distinguished when working over encrypted data:

- layers that can be evaluated homomorphically using leveled operations; and
- layers involving non-linear or more complex operations, in which case one or several programmable bootstrappings (PBS) are required.

We note that the first type of layers may also resort to bootstrap operations on some intermediate values whenever the noise exceeds a certain threshold.

5.1 Layers without PBS

Dense/linear layer. A (fully connected) dense layer computes the dot product between the inputs and a matrix of weights. A bias vector can be added. An activation function is then applied component-wise to produce the outputs. When there is no activation function, a dense layer is also called linear layer.

When evaluated homomorphically, the weights and the bias vector are provided in the clear. The evaluation of a dense layer (the activation excepted) thus consists of a series of multiplications by constants and additions, which are all leveled operations.

The activation functions are treated in the next section (see *activation layer*).

Convolution layer. A convolution layer convolves the input layer with a convolution kernel (a.k.a. filter) that is composed of a tensor of weights so as to produce a tensor of outputs. Biases can be added to the outputs. Moreover, an activation function can be applied to the outputs. Figure 7 illustrates a 2D convolution.

The filters are provided in the clear. Hence, as for the dense layer, the homomorphic evaluation of a convolution layer (without activation) consists of a series of multiplications by constants and additions.

Addition layer. An addition layer performs component-wise additions. Over encrypted data, those are leveled operations.

Flatten layer. A flatten layer reshapes its input into a lower-dimensional array so that it can for example be fed into a subsequent dense layer.

Over encrypted data, the flattening function simply consists in rearranging the input ciphertexts. No homomorphic operation is required.

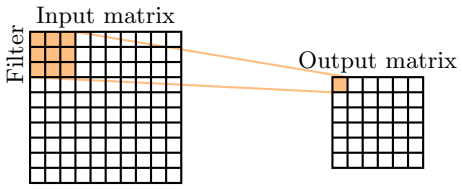


Fig. 7: Example of a 2D convolution.

Global average pooling layer. A global average pooling layer computes the average of the components of its inputs. Specifically, if n denotes the number of components and a_i denotes the value of component i, the global average pooling function computes $\left(\sum_{i=1}^{n} a_i\right)/n$.

In a homomorphic evaluation, the global average pooling can be reduced to the computation of the sum $\sum_{i=1}^{n} a_i$. The division by n is then performed in the next programmable bootstrapping—e.g., in a dense layer or a convolution layer—by dividing the weights by the same quantity. Hence, the sole homomorphic operation required to evaluate a global average pooling layer is the addition of ciphertexts, which is a leveled operation.

5.2 Layers with PBS

Activation layer: ReLU. An activation layer is used to inject non-linearity in the neural networks. It is crucial in the learning. There are many activation functions that can be used in an activation layer. One of the most popular activation functions is the Rectified Linear Unit (ReLU) function. Other commonly used activations include the sigmoid function or the hyperbolic tangent function.

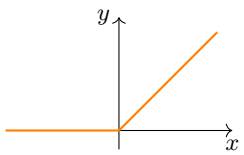


Fig. 8: ReLU function: $y = \max(0, x)$.

As detailed in Section 4, the homomorphic evaluation of an activation function (as any function) can be performed via a programmable bootstrapping (PBS), with the outputs of the function encoded inside the test polynomial.

Max-pooling layer. A max-pooling layer extracts a fixed-size subset of components from the inputs and computes their maximum.

At first sight, as the max function is multivariate (i.e., it takes multiple arguments on input), it is unclear how it can be evaluated homomorphically. With two arguments, the max function can however be expressed using the [univariate] ReLU function,

$$\max(x, y) = y + \mathrm{ReLU}(x - y) \ .$$

Hence, from $\mathsf{Encrypt}(x)$ and $\mathsf{Encrypt}(y)$, their maximum can be evaluated as $\mathsf{Encrypt}(\max(x, y)) = \mathsf{Encrypt}(y) + \mathsf{Encrypt}(\mathrm{ReLU}(z))$ with $\mathsf{Encrypt}(z) = \mathsf{Encrypt}(x) - \mathsf{Encrypt}(y)$. This requires a couple of addition/subtraction of ciphertexts plus the homomorphic evaluation of a ReLU function, the cost of which is one PBS.

In order to evaluate the max-pooling on more arguments, the basic relation $\max(x_1, \ldots, x_{k-1}, x_k) = \max(y_k, x_k)$ with $y_k = \max(x_1, \ldots, x_{k-1})$ can be used. For a series of k components $(x_1, \ldots, x_k)$, the homomorphic evaluation of the max-pooling function thus amounts to $(k-1)$ PBS.

6 Experimental Results & Benchmarks

We conducted a series of numerical experiments to assess the performance. We report below results against the MNIST dataset [24], which contains 28×28 images of handwritten digits. For testing purposes, we designed depth-20, 50, 100 neural networks, respectively noted NN-20, NN-50 and NN-100. These networks all include dense and convolution layers with activation functions; every hidden layer possesses at least 92 active neurons.

Parameter sets. The overall targeted security levels are 80 bits and 128 bits. The selected cryptographic parameters are defined by (k, N, σ) for $\overline{\text{GLWE}}$ encryption and (n, σ) for $\overline{\text{LWE}}$ encryption. The word-size is $\Omega = 64$ bits.

Table 1: Cryptographic parameters.

Security level		$\overline{\text{GLWE}}$			$\overline{\text{LWE}}$	
		k	N	σ	n	σ
80 bits	(I)	1	1024	2^{-40}	542	2^{-21}
	(II)	1	2048	2^{-60}	592	2^{-23}
128 bits	(III)	1	4096	2^{-62}	938	2^{-23}

The different parameter sets I, II & III meet at least the claimed security level and were validated using the `lwe-estimator` (https://bitbucket.org/malb/lwe-estimator/) [2]. They can be used for the homomorphic inference of networks requiring a maximal precision of 8 bits up to 12 bits.

Performance analysis. Experiments were performed on two different types of machines:

- a personal computer with 2.6 GHz 6-Core Intel® Core™ i7 processor, and
- a 3.00 GHz Intel® Xeon® Platinum 8275CL processor with 96 vCPUs hosted on AWS.

The two machines are referred to as PC and AWS.

For reference, we give in Table 2 the run-time and accuracy for an unencrypted inference. They were measured using `ONNX Runtime` [27]. The run-time and accuracy over encrypted data for different settings are presented in Table 3.

It is important to note that the given times correspond to the evaluation of a single inference run independently; in particular, the times are not amortized over a batch of inferences.

Table 2: Performance results in the clear.

	Run-time		Accuracy
	PC	AWS	
NN-20	0.17 ms	0.19 ms	97.5 %
NN-50	0.20 ms	0.30 ms	95.4 %
NN-100	0.33 ms	0.46 ms	95.2 %

The results of Table 3 clearly indicate the importance of the parameter choice and the different trade-offs that can be obtained. In particular, a larger value for parameter N increases the accuracy (at the expense of more processing).

Table 3: Performance results over encrypted data.

		Run-time		Accuracy
		PC	AWS	
80-*bit security*:				
NN-20	(I)	12.49 s	1.85 s	97.1 %
	(II)	30.04 s	5.10 s	**97.4 %**
NN-50	(I)	26.71 s	3.89 s	93.2 %
	(II)	71.71 s	10.66 s	**95.1 %**
NN-100	(I)	46.61 s	8.95 s	80.0 %
	(II)	108.73 s	21.05 s	**91.9 %**
128-*bit security*:				
NN-20	(III)	115.52 s	17.96 s	**97.5 %**
NN-50	(III)	233.55 s	37.69 s	**95.4 %**
NN-100	(III)	481.61 s	69.32 s	**90.5 %**

7 Conclusion

We presented a general framework for the evaluation of deep neural networks using fully homomorphic encryption. Our approach scales efficiently with the number of layers while providing good accuracy results. To do so we employ a versatile combination of encoding methods and of programmable bootstrapping techniques. To the best of our knowledge, our results set new records in the homomorphic inference of deep neural networks.

The practicality of our framework invites further works. First, it would be interesting to know the impact of the use of specialized hardware in our framework. We believe that several orders of magnitude in the processing times could be gained in that way. Another interesting work would be to investigate how to extend our techniques to the homomorphic training of neural networks or, more generally, to other intensive machine-learning tasks.

Availability

The library implementing our extended version of TFHE has been developed in Rust. It is available as an open-source project on GitHub at URL https://github.com/zama-ai/concrete.

References

1. Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data: A Short Course*. AMLbook.com, 2012. URL: http://amlbook.com.
2. Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015. doi:10.1515/jmc-2015-0016.
3. Marcelo Blatt, Alexander Gusev, Yuriy Polyakov, and Shafi Goldwasser. Secure large-scale genome-wide association studies using homomorphic encryption. Cryptology ePrint Archive, Report 2020/563, 2020. https://ia.cr/2020/563.
4. Joppe W. Bos, Kristin E. Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In *Cryptography and Coding (IMACC 2013)*, volume 8308 of *Lecture Notes in Computer Science*, pages 45–64. Springer, 2013. doi:10.1007/978-3-642-45239-0_4.
5. Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. Simulating homomorphic evaluation of deep learning predictions. In *Cyber Security Cryptography and Machine Learning (CSCML 2019)*, volume 11527 of *Lecture Notes in Computer Science*, pages 212–230. Springer, 2019. doi:10.1007/978-3-030-20951-3_20.

6. Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 483–512. Springer, 2018. doi:10.1007/978-3-319-96878-0_17.
7. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions Computation Theory*, 6(3):13:1–13:36, 2014. Earlier version in ITCS 2012. doi:10.1145/2633600.
8. Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. Classical hardness of learning with errors. In *45th Annual ACM Symposium on Theory of Computing*, pages 575–584. ACM Press, 2013. doi:10.1145/2488608.2488680.
9. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. *SIAM Journal on Computing*, 43(2):831–871, 2014. Earlier version in FOCS 2011. doi:10.1137/120868669.
10. California Consumer Privacy Act (CCPA). https://www.oag.ca.gov/privacy/ccpa.
11. Jung Hee Cheon and Damien Stehlé. Fully homomophic encryption over the integers revisited. In *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 513–536. Springer, 2015. doi:10.1007/978-3-662-46800-5_20.
12. 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. Earlier versions in ASIACRYPT 2016 and 2017. doi:10.1007/s00145-019-09319-x.
13. Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010. doi:10.1007/978-3-642-13190-5_2.
14. Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *33rd International Conference on Machine Learning (ICML 2016)*, volume 48 of *Proceedings of Machine Learning Research*, pages 201–210. PMLR, 2016. URL: http://proceedings.mlr.press/v48/gilad-bachrach16.html.
15. Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 617–640. Springer, 2015. doi:10.1007/978-3-662-46800-5_24.
16. The EU General Data Protection Regulation (GDPR). https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679&from=EN.
17. Craig Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 53(3):97–105, 2010. Earlier version in STOC 2009. doi:10.1145/1666420.1666444.
18. Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2013. doi:10.1007/978-3-642-40041-4_5.
19. Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer, 2009. doi:10.1007/978-0-387-84858-7.
20. iDASH secure genome analysis competition. http://www.humangenomeprivacy.org.
21. Miran Kim, Arif Harmanci, Jean-Philippe Bossuat, Sergiu Carpov, Jung Hee Cheon, Ilaria Chillotti, Wonhee Cho, David Froelicher, Nicolas Gama, Mariya Georgieva, Seungwan Hong, Jean-Pierre Hubaux, Duhyeong Kim, Kristin Lauter, Yiping Ma, Lucila Ohno-Machado, Heidi Sofia, Yongha Son, Yongsoo Song, Juan Troncoso-Pastoriza, and Xiaoqian Jiang. Ultra-fast homomorphic encryption models enable secure outsourcing of genotype imputation. bioXxiv, 2020. doi:10.1101/2020.07.02.183459.
22. Miran Kim, Yongsoo Song, Baiyu Li, and Daniele Micciancio. Semi-parallel logistic regression for GWAS on encrypted data. Cryptology ePrint Archive, Report 2019/294, 2019. https://ia.cr/2019/294.
23. Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015. doi:10.1007/s10623-014-9938-4.
24. Yann LeCun, Corinna Cortez, and Christopher C. J. Burges. The MNIST database of handwritten digits. Available at http://yann.lecun.com/exdb/mnist/.
25. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM*, 60(6):43:1–43:35, 2013. Earlier version in EUROCRYPT 2010. doi:10.1145/2535925.
26. Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 700–718. Springer, 2012. doi:10.1007/978-3-642-29011-4_41.
27. ONNX Runtime. Optimize and accelerate machine learning inferencing and training. https://microsoft.github.io/onnxruntime/index.html.

28. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):34:1–34:40, 2009. Earlier version in STOC 2005. doi:10.1145/1568318.1568324.
29. Ronald L. Rivest, Len Adleman, and Michael L. Detouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 165–179. Academic Press, 1978. Available at https://people.csail.mit.edu/rivest/pubs.html.
30. Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 617–635. Springer, 2009. doi:10.1007/978-3-642-10366-7_36.

A Appendix: $\overline{\text{LWE}}$ to $\overline{\text{LWE}}$ Conversion

We use the notations of Section 4.1.

Assume we are given the key-switching keys

$$\mathsf{ksk}[i,j] \leftarrow \overline{\text{LWE}}_{\boldsymbol{s}}\left(s'_i \cdot \tfrac{q}{B_{\mathsf{KS}}^j}\right) \quad \text{for all } 1 \le i \le kN \text{ and } 1 \le i \le \ell_{\mathsf{KS}}$$

for some parameters $B := B_{\mathsf{KS}}$ and $\ell := \ell_{\mathsf{KS}}$ defining a gadget decomposition (see Section 3.3). Adapting [12, § 4.1] teaches that, on input $\overline{\text{LWE}}$ ciphertext $\boldsymbol{c'} \leftarrow \overline{\text{LWE}}_{\boldsymbol{s'}}(\bar{\mu}) = (\bar{a}'_1, \ldots, \bar{a}'_{kN}, \bar{b}') \in (\mathbb{Z}/q\mathbb{Z})^{kN+1}$ under the key $\boldsymbol{s'} = (s_1, \ldots, s_{kN}) \in \mathbb{B}^{kn}$,

$$\bar{\boldsymbol{c}}'' := (0, \ldots, 0, \bar{b}') - \sum_{i=1}^{kN} \sum_{j=1}^{\ell_{\mathsf{KS}}} \bar{a}'_{i,j}\, \mathsf{ksk}[i,j] \in (\mathbb{Z}/q\mathbb{Z})^{n+1}$$

where

$$(\bar{a}'_{i,1}, \ldots, \bar{a}'_{i,\ell_{\mathsf{KS}}}) = \boldsymbol{g}^{-1}(\bar{a}'_i)$$

is a $\overline{\text{LWE}}$ encryption of $\bar{\mu}$ under key $\boldsymbol{s}$, provided that the resulting noise keeps small.