CONCRETE: Concrete Operates oN Ciphertexts Rapidly by Extending TfhE

Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, Samuel Tap
Zama, France
https://zama.ai

ABSTRACT

Fully homomorphic encryption (FHE) extends traditional encryption schemes. It allows one to directly compute on encrypted data without requiring access to the decryption key. This paper introduces CONCRETE, an open-source library developed in Rust that builds on the state-of-art TFHE cryptosystem. It provides a user-friendly interface making FHE easy to integrate. The library deals with inputs of arbitrary format and comes with an extensive set of operations to play with ciphertexts, including a programmable bootstrapping. CONCRETE is available on GitHub at URL https://github.com/zama-ai/concrete and on https://crates.io.

1 INTRODUCTION

The idea of fully homomorphic encryption (FHE) emerged in 1978 when Rivest, Adleman and Dertouzos started talking about privacy homomorphisms [9]. However, no solution was found until 2009 when Gentry presented the first instantiation of a FHE scheme [5].

A common feature in Gentry's original cryptosystem and in all subsequent FHE schemes is that ciphertexts contain *noise*. The vast majority of homomorphic operations make this noise grow. If not controlled, the noise in a ciphertext can compromise the encrypted plaintext and induce incorrect results at decryption time. This fact inherently limits the number of operations that can be performed on ciphertexts. A homomorphic encryption scheme supporting a predetermined noise threshold is termed *leveled*. The groundbreaking idea of [5] which made FHE possible, is a technique called *bootstrapping* enabling to reduce the noise—using only public material—when needed. This way, there is no more limitation on the maximal number of operations that can be performed and the scheme becomes *fully* homomorphic.

Since Gentry's FHE scheme, many other schemes have been presented in the literature, but the bootstrapping technique has remained the main bottleneck, both in terms of running time (several minutes) and of key sizes (gigabytes). This state of affairs dramatically changed in 2015 when Ducas and Micciancio introduced the FHEW cryptosystem [3] mastering a bootstrapped NAND gate in less than a second yet with keys of about 1 GB in size. This was further improved in 2016 by Chillotti *et al.* and gave rise to the TFHE cryptosystem [2] running any bootstrapped binary gate in a few tens of milliseconds and with keys in the order of megabytes.

Furthermore, in addition of being relatively fast, the bootstrapping procedure in TFHE can be 'programmed': it enables the homomorphic evaluation of any function on a ciphertext while reducing the noise. This powerful technique opens new application scenarios for the practical use of FHE technologies.

The above reasons explain our choice to implement and extend the TFHE scheme in our CONCRETE library (CONCRETE stands for "Concrete Operates oN Ciphertexts Rapidly by Extending TfhE"). As for the programming language, we picked *Rust*, a new language that has become popular thanks to its performance (as good as C or C++ with the advantages of a high-level programming language), its memory safety, and its efficient memory usage and access.

CONCRETE follows TFHE's implementation: it makes use of arithmetic modulo q with a small modulus natural for a machine such as $q = 2^{32}$ or $q = 2^{64}$. This is very convenient because these moduli are natively supported while using unsigned integers with 32 or 64 bits and their respective efficient addition and multiplication operations. Further, CONCRETE unleashes the full potential of TFHE by encrypting and bootstrapping values of arbitrary format, including real numbers. In particular, it is not limited to boolean computation. This is achieved through the use of encoding functions.

The library is designed in layers. The lower-level layer, called *Core API*, is meant to be accessible by FHE experts and aims to be as efficient as possible. The layer above, named *Crypto API*, wraps the operator layer in a user-friendly way. It is accessible and easy to use by any programmer (even with a limited understanding of the cryptography behind it). It offers in addition extra capabilities, including automated *metadata* keeping track of the amount of noise.

2 CRYPTOGRAPHIC CONTENT OF THE LIBRARY

TFHE [2] is based on the *learning with errors* (LWE) problem [8], a hard problem on lattices, and its variant *Ring-LWE* [7, 10]. It deals with two types of ciphertexts: LWE ciphertexts and RLWE ciphertexts.

2.1 Encryption & Decryption

LWE *ciphertexts.* Let $\mathbb{B}$ denote the set $\{0, 1\}$ and let q be a power of two. Let also $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$. The plaintext space is $\mathbb{Z}_q$. An LWE ciphertext $\boldsymbol{c} \leftarrow \mathrm{LWE}_{\boldsymbol{s}}(\mu)$ encrypting a plaintext $\mu \in \mathbb{Z}_q$ under the secret key $\boldsymbol{s} = (s_1, \dots, s_n) \in \mathbb{B}^n$ is a tuple

$$\boldsymbol{c} = (a_1, \dots, a_n, b) \in (\mathbb{Z}_q)^{n+1}$$

where $a_1, \dots, a_n$ are sampled uniformly at random in $\mathbb{Z}_q$ and $b = \sum_{j=1}^{n} s_j\, a_j + \mu + e \pmod q$, with e a (small) discretized Gaussian noise.

Given a ciphertext $(a_1, \dots, a_n, b)$, the decryption algorithm uses the secret key $(s_1, \dots, s_n)$ to obtain $\mu^* := b - \sum_{j=1}^{n} s_j\, a_j = \mu + e \pmod q$; see Fig. 1.3. If the message is encoded in the most significant bits (and thus if μ has its least significant bits set to zero) then, provided that the noise e is not too large, rounding μ^* enables to recover the original μ.

RLWE *ciphertexts.* Let N be a power of two so that $X^N + 1$ is the $2N$th cyclotomic polynomial. Let also $\mathbb{B}_N[X] = \mathbb{B}[X]/(X^N + 1)$

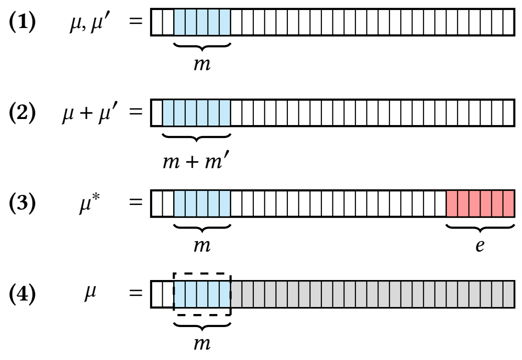

Figure 1: Representation of plaintexts as a sequence of bits. White boxes indicate bits set to zero; MSBs are on the left and LSBs on the right.

and $\mathbb{Z}_{q,N}[X] = \mathbb{Z}_q[X]/(X^N + 1)$. The plaintext space for RLWE encryption is $\mathbb{Z}_{q,N}[X]$. An RLWE ciphertext $\boldsymbol{\mathfrak{c}} \leftarrow \mathrm{RLWE}_{\boldsymbol{\mathfrak{s}}}(\upmu(X))$, encrypting a plaintext $\upmu(X) \in \mathbb{Z}_{q,N}[X]$ under the secret key $\boldsymbol{\mathfrak{s}} = (\mathfrak{s}_1(X), \dots, \mathfrak{s}_k(X)) \in (\mathbb{B}_N[X])^k$ is a tuple

$$\boldsymbol{\mathfrak{c}} = (\mathfrak{a}_1(X), \dots, \mathfrak{a}_k(X), \mathfrak{b}(X)) \in (\mathbb{Z}_{q,N}[X])^{k+1}$$

of $k + 1$ polynomials, where $\mathfrak{a}_1(X), \dots, \mathfrak{a}_k(X)$ are sampled uniformly at random in $\mathbb{Z}_{q,N}[X]$ and $\mathfrak{b}(X) = \sum_{j=1}^{k} \mathfrak{s}_j(X)\,\mathfrak{a}_j(X) + \upmu(X) + \mathfrak{e}(X) \pmod{(q, X^N+1)}$, with $\mathfrak{e}(X)$ a polynomial with (small) discretized Gaussian coefficients.

Given a ciphertext $(\mathfrak{a}_1(X), \dots, \mathfrak{a}_k(X), \mathfrak{b}(X))$, the decryption proceeds by computing (in $\mathbb{Z}_{q,N}[X]$) the polynomial $\upmu^*(X) := \mathfrak{b}(X) - \sum_{j=1}^{k} \mathfrak{s}_j(X)\,\mathfrak{a}_j(X) = \upmu(X) + \mathfrak{e}(X)$. As for LWE ciphertexts, if the message is encoded in the most significant bits of the coefficients of $\upmu(X)$, rounding each coefficient of $\upmu^*(X)$ yields back the original $\upmu(X)$, provided that the noise $\mathfrak{e}(X)$ is not too large.

2.2 Real Encoding on $[a, b]$

The plaintext space is defined over $\mathbb{Z}_q$. This section describes how to represent messages of a real domain $\mathfrak{D} = [a, b] \subset \mathbb{R}$ with a subset of $\mathbb{Z}_q$. To do so, a number of bits n_{msg} are reserved to store the message $m \in \mathfrak{D}$ as well as a number of padding bits n_{pad} in the MSBs. There are two different settings for the encoding: (i) the number of precision bits contained in the plaintext is maximized (Fig. 1.4) and (ii) the precision is limited to n_{msg} bits (Fig. 1.1);

The encoding functions are defined as (i) $\mathsf{Encode}(m \in \mathfrak{D}) = \lceil q\,(m-a)/(\delta \cdot 2^{n_{\mathrm{pad}}}) \rfloor$ and as (ii) $\mathsf{Encode}(m \in \mathfrak{D}) = \frac{q}{2^{n_{\mathrm{msg}}+n_{\mathrm{pad}}}} \lceil (m-a) \cdot 2^{n_{\mathrm{msg}}}/\delta \rfloor$ with $\delta = b - a + \sigma$ and $\sigma = (b-a)/(2^{n_{\mathrm{msg}}} - 1)$. The quantity σ is a security margin to prevent wrong decryption nearby $0 \in \mathbb{Z}_q$ (i.e., a and b in $\mathfrak{D}$), which boils down to actually consider the interval $[a, b + \sigma)$.

The decoding functions of a noisy plaintext $\mu^* \in \mathbb{Z}_q$ are defined as (i) rounding μ^* to remove the noise and get μ and then computing $\mathsf{Decode}(\mu) = a + (\mu \cdot \delta \cdot 2^{n_{\mathrm{pad}}}/q) \in \mathfrak{D}$, and (ii) simply computing $\mathsf{Decode}(\mu^*) = a + (\mu^* \cdot \delta \cdot 2^{n_{\mathrm{pad}}}/q) \in \mathfrak{D}$.

2.3 Addition & Scalar Multiplication

LWE ciphertexts can readily be added together (component-wise addition). The result of an addition is an LWE ciphertext in $(\mathbb{Z}_q)^{n+1}$ encrypting the sum (modulo q) of the input plaintexts.

This works analogously with RLWE ciphertexts. Note that in both cases the noises also add up as a result of the addition.

Remark 1. A way to avoid the modular reduction in the sum of the plaintexts is to make sure that they are small enough. For example, suppose that $q = 2^{32}$ and that plaintexts viewed as 32-bit integers are of the form $\{x \cdot 2^{25}, 0 \le x < 2^5\}$. This means that plaintexts are made of 2 bits set to zero (i.e., the 2 left-most significant bits—these bits are called *padding bits*), 5 bits containing the message, and 25 bits set to zero; see Fig. 1.1. Hence, as illustrated in Fig. 1.2, the addition of two such plaintexts yields a plaintext as an element of $\{x \cdot 2^{25}, 0 \le x < 2^6\}$, that is, with only one bit of padding left.

By extension, LWE (resp. RLWE) ciphertexts can be multiplied by an integer constant. The noise after the scalar multiplication grows proportionally with respect to the integer constant.

2.4 External Product

LWE and RLWE ciphertexts can be added; unfortunately, they cannot be multiplied as easily. One may resort to a matrix-based approach [6] to this end. This involves two parameters: a basis $B \in \mathbb{N}$ and a number of levels $\ell \in \mathbb{N}$. To avoid rounding, it is assumed that $B^\ell \mid q$.

RGSW ciphertexts. An RGSW ciphertext of $\upmu(X) \in \mathbb{Z}_N[X]$ under secret key $(\mathfrak{s}_1(X), \dots, \mathfrak{s}_k(X)) \in \mathbb{B}_N[X]^k$ is a matrix $\boldsymbol{\mathfrak{C}} \in (\mathbb{Z}_{q,N}[X])^{(k+1)\ell \times (k+1)}$ whose row #t where $t = i + (j-1)\ell$, for $1 \le i \le \ell$ and $1 \le j \le k+1$, is an RLWE encryption under key $(\mathfrak{s}_1(X), \dots, \mathfrak{s}_k(X))$ of plaintext $\upmu_t(X) := -\mathfrak{s}_j(X)\,\upmu(X)\,\frac{q}{B^i}$, with $\mathfrak{s}_{k+1}(X) := -1$. This is written $\boldsymbol{\mathfrak{C}} \leftarrow \mathrm{RGSW}_{\boldsymbol{\mathfrak{s}}}(\upmu(X))$.

External product. The external product [2, § 3.3] of an RGSW ciphertext $\boldsymbol{\mathfrak{C}_1} \leftarrow \mathrm{RGSW}_{\boldsymbol{\mathfrak{s}}}(\upmu_1(X))$ and of an RLWE ciphertext $\boldsymbol{\mathfrak{c}_2} \leftarrow \mathrm{RLWE}_{\boldsymbol{\mathfrak{s}}}(\upmu_2(X))$ is noted $\boldsymbol{\mathfrak{c}_3} = \boldsymbol{\mathfrak{C}_1} \boxdot \boldsymbol{\mathfrak{c}_2}$. The resulting ciphertext $\boldsymbol{\mathfrak{c}_3}$ is an RLWE encryption of $\upmu_1(X) \cdot \upmu_2(X)$. The external product is defined by $\boldsymbol{\mathfrak{C}_1} \boxdot \boldsymbol{\mathfrak{c}_2} = \mathbf{G}^{-1}(\boldsymbol{\mathfrak{c}_2}) \cdot \boldsymbol{\mathfrak{C}_1}$ where $\mathbf{G}^{-1}(\boldsymbol{\mathfrak{c}_2})$ is the gadget decomposition of $\boldsymbol{\mathfrak{c}_2}$. The transformation $\mathbf{G}^{-1}(\cdot)$ flattens a vector of $(k+1)$ polynomials of $\mathbb{Z}_{q,N}[X]$ into a row vector of $(k+1)\ell$ polynomials of $\mathbb{Z}_N[X]$ with small coefficients in the range $[-B/2, B/2)$. The use of the gadget decomposition enables to control the noise growth in the external product.

CMUX. The main application of the external product in TFHE is the controlled multiplexer, or CMUX in short. Given two RLWE ciphertexts $\boldsymbol{\mathfrak{c}_0} \leftarrow \mathrm{RLWE}_{\boldsymbol{\mathfrak{s}}}(\upmu_0(X))$ and $\boldsymbol{\mathfrak{c}_1} \leftarrow \mathrm{RLWE}_{\boldsymbol{\mathfrak{s}}}(\upmu_1(X))$, the CMUX operator acts as a selector to choose between $\boldsymbol{\mathfrak{c}_0}$ and $\boldsymbol{\mathfrak{c}_1}$ according to an RGSW encryption $\boldsymbol{\mathfrak{C}_\mathrm{b}} \leftarrow \mathrm{RGSW}_{\boldsymbol{\mathfrak{s}}}(\mathrm{b})$ of a control bit $\mathrm{b} \in \mathbb{B}$. This can be computed through an external product as $\mathrm{CMUX}(\boldsymbol{\mathfrak{C}_\mathrm{b}}, \boldsymbol{\mathfrak{c}_0}, \boldsymbol{\mathfrak{c}_1}) = \boldsymbol{\mathfrak{C}_\mathrm{b}} \boxdot (\boldsymbol{\mathfrak{c}_1} - \boldsymbol{\mathfrak{c}_0}) + \boldsymbol{\mathfrak{c}_0}$. The output is an RLWE encryption of $\upmu_\mathrm{b}(X)$.

2.5 Programmable Bootstrapping

As aforementioned, homomorphic operations may increase the noise in ciphertexts, and when the noise grows too much ciphertexts cannot be decrypted anymore. This is why, from time to time, it is important to refresh noisy ciphertexts by reducing the amount of noise. *Programmable bootstrapping* (PBS) is a generalization of the bootstrapping technique allowing resetting the noise at a fixed level while at the same time homomorphically evaluating a function on the input ciphertext.

Programmable bootstrapping involves a series of three algorithms: (i) Switch Modulus, (ii) Blind Rotate, and (iii) Sample Extract. It is often followed by an LWE Key Switching algorithm. These algorithms are described below.

Modulus switching. The modulus switching is a classical technique in FHE for switching the modulus of ciphertexts. In TFHE, a modulus switching is performed prior to the blind rotation procedure. The Switch Modulus algorithm scales by $2N/q$ and rounds (i.e., performs the operation $\lceil 2N\,(\,\cdot\mod q)/q\rfloor$) each component of the input LWE ciphertext (i.e., elements in $\mathbb{Z}_q$).

Blind rotation. The Blind Rotate algorithm homomorphically applies a rotation over the coefficients of the plaintext polynomial stored in an RLWE ciphertext. It is used to blindly put a desired coefficient in the constant term. This algorithm takes on input a (possibly trivial)[1] RLWE ciphertext $\boldsymbol{c}_T \in (\mathbb{Z}_{q,N}[X])^{k+1}$ encrypting the test polynomial $T(X) = t_0 + t_1 X + \cdots + t_{N-1} X^{N-1}$, an LWE ciphertext $\boldsymbol{c} = (a_1, \ldots, a_n, b) \in (\mathbb{Z}_{2N})^{n+1}$ encrypting μ, and n RGSW ciphertexts $\{\mathcal{C}_i\}_{1\le i\le n}$ encrypting each bit of the secret key $(s_1, \ldots, s_n) \in \mathbb{B}^n$ used to encrypt $\boldsymbol{c}$. Note that the RGSW ciphertexts and the RLWE ciphertext are encrypted with the same secret key. The noisy plaintext contained in $\boldsymbol{c}$ is noted μ^*.

In Figure 2, the test polynomial is seen as a table where, for $0 \le i \le N-1$, $t[i]$ represents the i-th coefficient t_i of $T(X)$, and $t[i+N] = -t_i$. The algorithm starts by initializing an accumulator $\mathrm{ACC} \leftarrow X^{-b} \cdot \boldsymbol{c}_T$. Next, by using each ciphertext of $\{\mathcal{C}_j\}_{1\le j\le n}$ as control bit into a CMUX, the algorithm successively updates the accumulator as $\mathrm{ACC} \leftarrow \mathsf{CMUX}(\mathcal{C}_j, \mathrm{ACC}, X^{a_j} \cdot \mathrm{ACC})$ for $j = 1, \ldots, n$. The output is an RLWE ciphertext encrypting the plaintext $X^{-b+\sum_{j=1}^{n} s_j\, a_j} \cdot \boldsymbol{c}_T = X^{-\mu^*} \cdot \boldsymbol{c}_T$ which contains $t[\mu^*]$ in its constant coefficient.

Sample extraction. It is easy to extract any of the N coefficients inside an RLWE plaintext $\mu(X) = \mu_0 + \mu_1 X + \cdots + \mu_{N-1} X^{N-1} \in \mathbb{Z}_{q,N}[X]$ as an LWE ciphertext. The corresponding algorithm is called Sample Extract and simply consists in picking some of the coefficients of the input RLWE ciphertext and using them to build the output as an LWE encryption of the desired coefficient μ_i. It is a public operation and adds no noise in the output.

Key switching. After the blind rotation and sample extraction, the resulting LWE ciphertext is encrypted with a secret key that is different from the one used on input. In order to revert to the original key, one needs to perform a key-switching operation (this operation is optional). As for the external product, the corresponding Key Switch algorithm involves two parameters: a basis $B_{\text{KS}} \in \mathbb{N}$ and a number of levels $\ell_{\text{KS}} \in \mathbb{N}$. The input dimension is denoted n_{in} and the output dimension n_{out}. This procedure is classical in FHE and is possible thanks to the use of a key-switching key; i.e., a list of n_{in} LWE ciphertexts encrypting each bit of the input secret key multiplied by successive powers of basis B_{KS} (up to ℓ_{KS}). The output is an LWE ciphertext encrypting the same message as the input, with respect to the original secret key.

[1]A trivial RLWE ciphertext for plaintext $\mu(X)$ is a ciphertext of the form $(0, \ldots, 0, \mu(X))$.

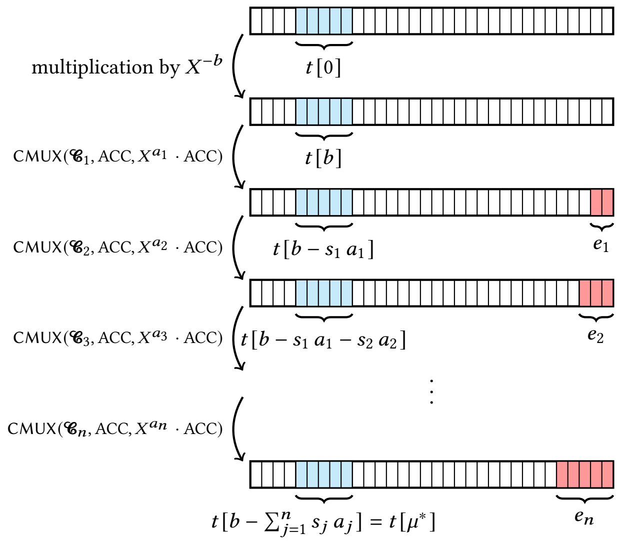
multiplication by X^{-b}
$t[0]$
$\mathsf{CMUX}(\mathcal{C}_1, \mathrm{ACC}, X^{a_1} \cdot \mathrm{ACC})$
$t[b]$
$\mathsf{CMUX}(\mathcal{C}_2, \mathrm{ACC}, X^{a_2} \cdot \mathrm{ACC})$
$t[b - s_1\, a_1]$
e_1
$\mathsf{CMUX}(\mathcal{C}_3, \mathrm{ACC}, X^{a_3} \cdot \mathrm{ACC})$
$t[b - s_1\, a_1 - s_2\, a_2]$
e_2
$\mathsf{CMUX}(\mathcal{C}_n, \mathrm{ACC}, X^{a_n} \cdot \mathrm{ACC})$
$t[b - \sum_{j=1}^{n} s_j\, a_j] = t[\mu^*]$
e_n

Figure 2: Binary representation of the constant coefficient from the polynomial plaintext encrypted in the accumulator during the Blind Rotate algorithm.

Programmable Bootstrapping. The programmable bootstrapping algorithm implemented in CONCRETE takes on input an LWE ciphertext $\boldsymbol{c}_{\text{in}}$ encrypting a plaintext μ_{in} under the key $\boldsymbol{s}_{\text{in}} \in \mathbb{B}^n$, and a bootstrapping key; i.e., a list of n RGSW ciphertexts, each one encrypting a bit of the secret key $\boldsymbol{s}_{\text{in}}$ used to encrypt $\boldsymbol{c}_{\text{in}}$. It outputs an LWE ciphertext $\boldsymbol{c}_{\text{out}}$ encrypting the plaintext μ_{out} under the key $\boldsymbol{s}_{\text{out}} \in \mathbb{B}^{kN}$.

As depicted in Figure 3, this procedure is composed of three steps:

(1) a modulus switching (to be compatible with N) wherein the input ciphertext in $(\mathbb{Z}_q)^{n+1}$ is converted into a ciphertext in $(\mathbb{Z}_{2N})^{n+1}$;
(2) a blind rotation of an RLWE ciphertext encrypting a redundant look-up table t of size N (which is rotated according to the output LWE ciphertext of Step 1);
(3) a sample extraction of the constant coefficient of the RLWE.

It is essential to have at least one bit of padding left in the input LWE plaintext. This is a consequence of the negacyclic property: $X^{N+u} = -X^u$. Without padding the output is only guaranteed up to the sign.

Look-up table. There are multiple ways to represent a function $g\colon \mathbb{Z}_N \to \mathbb{Z}_q$. A simple way is to use a look-up table, which is a sort of dictionary containing pairs $(i, t[i])$, where i is the input index and $t[i] = g(i)$. In our implementation, the value $t[i]$ is obtained by performing $\mathsf{Encode}' \circ f \circ \mathsf{Decode}(i)$. Observe that the encoding function might not be the one corresponding to Decode (i.e., $\mathsf{Encode}' \neq \mathsf{Encode}$): in fact, the bootstrapping enables to change encoding. These values are used to program the test polynomial

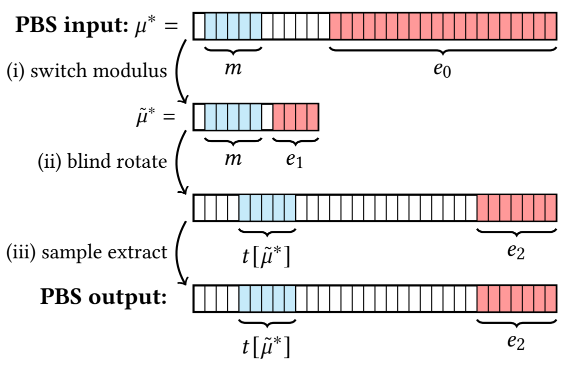

Figure 3: Representation of the PBS steps as a sequence of bits. Each rectangle shows the noisy plaintext encrypted by an LWE ciphertext but the third which is the plaintext in the constant coefficient of an RLWE ciphertext.

as $T(X) = t_0 + \cdots + t_{N-1}X^{N-1}$ with $t_i = t[i]$, and store it in an RLWE ciphertext. It is easy to see that the constant coefficient of $T(X) \cdot X^{-i}$ is $t[i]$, that is, the right output in the look-up table.

Note that if the goal of the bootstrapping is simply to reduce the noise, function f is the identity function.

Computing a Multiplication. Interestingly, a multiplication can be computed using two PBS operations. This follows from the observation that, in $\mathbb{R}$, $x \cdot y = (x+y)^2/4 - (x-y)^2/4$. Over encrypted data, this amounts to one homomorphic addition and one homomorphic subtraction to get the evaluation of $x+y$ and $x-y$, two PBS's with function $z \mapsto z^2/4$ to get $(x+y)^2/4$ and $(x-y)^2/4$, and finally one homomorphic subtraction to get $x \cdot y$.

3 CONCRETE IMPLEMENTATION

The CONCRETE[2] library is built as a stack of layers (Fig. 4) from the closest to the hardware to the most abstracted and easy-to-use (Crypto API). In between, the API called Core API proposes an hardware generic API. At the moment, Core API only provides access to the CPU-based code version, but a GPU version (following the same API) is in preparation. On top of that, the Core API is wrapped in a more user-friendly API called Crypto API where every cryptographic object is represented as a structure. These structures also contain some metadata as the noise distribution, the number of padding bits or the decoding parameters. They are mainly used to ensure the correctness of the computation. The noise is monitored by a dedicated module abbreviated as NPE, for Noise Propagation Estimator.

The value of q is not fixed so that all functions are currently defined for either 2^{32} or 2^{64}. Note that having functions for $q = 2^{128}$ would be easy.

```
use concrete_lib::*;

fn main() -> Result<(), CryptoAPIError> {
    // parameters
    let (min, max) = (-1., 1.);
    let weight = 2.;
    let bias = 3.;

```

[2]The CONCRETE library is released under the **AGPLv3** license.

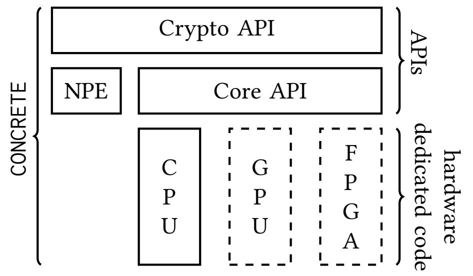

Figure 4: Overview of the CONCRETE library architecture. Dashed boxes refer to upcoming features.

```
    // message
    let message = 0.4;

    // input encoder
    let encoder_in = Encoder::new(min, max, 6, 7)?;

    // secret keys
    let rlwe_sk = RLWESecretKey::new(&RLWE128_1024_1);
    let lwe_sk = rlwe_sk.to_lwe_secret_key();

    // bootstrapping key
    let bsk = LWEBSK::new(&lwe_sk, &rlwe_sk, 7, 3);

    // encode and encrypt (x0, x1)
    let ct_in = LWE::encode_encrypt(&lwe_sk, message, &encoder_in)?;

    // ct_x <- ct_in * weight
    let ct_x = ct_in.mul_constant_with_padding(weight, 3., 8)?;

    // output encoder
    let max_out = weight * max + bias;
    let encoder_out = Encoder::new(0., max_out, 6, 2)?;

    // ct_y <- ct_x + b
    let ct_y = ct_x.add_constant_dynamic_encoder(bias)?;

    // ct_res <- max(0., ct_y)
    let ct_res = ct_y.bootstrap_with_function(&bsk,
                    |x| f64::max(x, 0.), &encoder_out)?;

    // decrypt
    let res = ct_res.decrypt_decode(&lwe_sk)?;
    Ok(())
}
```

Code Example 1: A piece of code using CONCRETE's Crypto API to compute a multiplication with a constant, an addition with a constant, a PBS with the ReLU function and to decrypt.

NPE. In a nutshell, the Noise Propagation Estimator (NPE) contains the noise formula associated to each homomorphic operator. It uses parts of the metadata associated to the ciphertext as inputs; i.e., the variances of the noise distribution. In the end, the noise variance of the output ciphertext is updated. In the case of computation with a cleartext (e.g., between a scalar and a ciphertext), the actual scalar is required as input.

FFT. The Fast Fourier Transform (FFT) is used to speed up the polynomial products, as provided by the FFTW [4] implementation. The latter has been slightly modified to be compliant with the

$\mathbb{Z}_{q,N}[X]$ arithmetic. The idea is to multiply each polynomial coefficient by power of $e^{i\pi/N}$ for the forward FFT and by power of $e^{-i\pi/N}$ for the backward FFT. The Fourier representation of the polynomials uses 64-bit floating numbers.

3.1 `Core API`

The `Core API` is designed as an abstraction of the hardware dedicated code. Then depending on the underlying hardware actually present on the running machine, the API offers multiple choices which do not require any code modification by the end-user. For instance, the `Core API` offers some SIMD optimizations if supported by the CPU, or even running on a GPU if possible. Note that the latter is currently a mock-up and not open-sourced yet. Despite this convenient abstraction, the API aims to be as low-level as possible in order to yield better performance.

The API is divided into two main modules: (i) *math* containing non-cryptographic operations such as adding, subtracting or multiplying (unsigned) integers or polynomials, and (ii) *crypto* containing cryptographic-related operations such as encryption and decryption functions or bootstrapping and key switching functions.

Slices are used to represent messages, plaintexts, ciphertexts and keys. In Rust, a slice is a contiguous sequence of elements but at the difference of an array, its size is not known at compile time. A slice contains a pointer to the data and the length of the slice. Messages are stored in slices of `f64`, and plaintexts, ciphertexts and keys are stored in slices of unsigned integers `u32` or `u64`.

Ciphertexts. Ciphertexts are stored in slices of unsigned 32-bit/64-bit integers. In `Core API`, functions are able to work on a collection of ciphertexts. This is modeled as a single slice containing several concatenated LWE ciphertexts. For instance, the key switch function can compute a key switch for each of the ciphertexts in the slice.

Secret keys. The LWE or RLWE secret keys are all sampled from a uniform binary distribution. The keys are stored in the same way as ciphertexts: in slices of unsigned 32-bit/64-bit integers such that in the first element of the slice are the 32/64 first bits of the key.

Random source. Uniform random integers are generated with *OpenSSL*. Discretized Gaussian integers are obtained with a Box–Muller algorithm using OpenSSL uniform random integers.

Benchmarks. Tables 1 and 2 show the time needed to compute a PBS and a key switch for both $q = 32$ and $q = 64$. Benchmarks are obtained using Criterion, a Rust micro-benchmarking library. Each measurement is the mean duration of 500 iterations of the operation performed on a single thread.

Benchmarks were made on a personal computer with a 2.6 GHz 6-Core Intel® Core™ i7 processor.

3.2 `Crypto API`

The `Crypto API` wraps `Core API` in simple-to-use structures. Typically, metadata are directly included and automatically updated as the `Core API` homomorphic computation progresses. Code example 1 shows how to pick cryptographic parameters and generate secret keys (Lines 16 and 17), how to define an input encoder (Line 13), how to encode and encrypt a message (Line 23), how to generate a bootstrapping key (Line 20), how to define an output encoder (Line 30), how to homomorphically compute $c \to \max(0., weight{\cdot}c{+}bias)$ with a PBS (Lines 36 and 37), and finally how to decrypt (Line 40). Code example 2 shows how to pick cryptographic parameters and how to generate an RLWE secret key (Lines 2 and 3), how to define an input encoder (Line 6), how to encode and encrypt packed messages in RLWE ciphertexts (Lines 7 to 10), how to add two ciphertexts with padding (Line 13), to multiply by constants with padding (Line 14), how to extract an LWE from an RLWE (Line 16), and finally how to decrypt (Line 18).

Table 1: PBS in milliseconds from CPU `Core API` with $k = 1$, $B = 2^7$ and $\ell = 3$.

	$N = 1024$		$N = 2048$		$N = 4096$	
# bits	32	64	32	64	32	64
$n = 630$	15.49	18.08	33.28	39.54	73.22	84.01
$n = 800$	19.23	22.98	42.33	50.53	93.12	107.3
$n = 1024$	24.54	29.16	54.14	64.18	117.9	135.2

Table 2: LWE key switching (in milliseconds) from CPU `Core API` with $B_{\text{KS}} = 4$ and $\ell_{\text{KS}} = 8$.

	$n_{\text{out}} = 630$		$n_{\text{out}} = 800$		$n_{\text{out}} = 1024$	
# bits	32	64	32	64	32	64
$n_{\text{in}} = 1024$	0.9912	2.292	1.171	2.872	1.518	3.652
$n_{\text{in}} = 2048$	2.130	4.661	2.495	5.877	3.200	7.699
$n_{\text{in}} = 4096$	4.413	9.476	5.134	11.89	6.592	15.39

Structures. The `Crypto API` introduces the following structures: `Encoder`, `Plaintext`, `LWESecretKey`, `RLWESecretKey`, `LWEBSK`, `LWEKSK`, `LWE`, `VectorLWE`, `VectorRLWE`, `LWEParams`, `RLWEParams`.

Cryptographic parameters. In order to ease the use of the library, some secure cryptographic parameters are offered to the end-user. For instance, in Code example 1, an RLWE secret key is created using `&RLWE128_1024_1`. Following the notations defined in Section 2, this means that the RLWE key provides 128 bits of security with $N = 1024$ and $k = 1$. Similar predefined sets are available for LWE parameters, such as `&LWE128_630` providing 128 bits of security with $n = 630$. The adequate standard noise distribution for each parameter set have been computed via the LWE Estimator [1] in September 2020. Obviously for more advanced users, the cryptographic parameters can easily be set by hand.

```
1  // generation of a RLWE secret key
2  let rlwe_params = RLWEParams::new(1024, 1, −25)?;
3  let rlwe_sk = RLWESecretKey::new(&rlwe_params);
4
5  // encryption
6  let encoder = Encoder::new(−5., 5., 5, 7)?;
7  let rlwe_ct_1 = VectorRLWE::encode_encrypt_packed(
8      &rlwe_sk, &[1.4, −3.5, 2.7], &encoder)?;
9  let rlwe_ct_2 = VectorRLWE::encode_encrypt_packed(
10     &rlwe_sk, &[−3.1, −2.2, 0.3], &encoder)?;
11
12 // homomorphic operations
13 let mut rlwe_ct_3 = rlwe_ct_1.add_with_padding(&rlwe_ct_2)?;
14 rlwe_ct_3.mul_constant_with_padding_inplace(&[2.3, 2., −1.5], 2.3, 5)?;
```

```
let lwe_ct = rlwe_ct_3.extract_1_lwe(1,     0)?;

// decryption
let lwe_sk = rlwe_sk.to_lwe_secret_key();
let d = lwe_ct.decrypt_decode(&lwe_sk)?;
```

Code Example 2: A piece of code using `CONCRETE`'s `Crypto API` to encrypt packed messages with RLWE and to compute constant multiplication and homomorphic addition before extracting one slot as an LWE ciphertext.

Encoders and ciphertexts. An encoder is a structure containing the bounds of the real interval, the number of bits of precision, the number of padding bits and the type of encoding. The ciphertext structures `VectorLWE` and `VectorRLWE` defined in the `Crypto API` are able to store one or more ciphertexts of the same type (namely LWE or RLWE). It also contains the required metadata to track the noise, the encoding, and the number of bits of message guaranteed to be correct all along computations. The noise evaluation is realized with the help of the `NPE` module. In the case where the noise bits are devouring the bits of the message, a message informs the user about the intact number of bits and the number of bits of message is updated.

To encrypt, a call to the function `LWE::encode_encrypt` for instance, allows one to obtain an LWE ciphertext from a message, a secret key and an encoder. Note that the same function is present in modules `VectorLWE` and `VectorRLWE`.

Padding bits. Some homomorphic functions require some bits of padding in order to avoid the reduction modulo q and loose some most significant bits of the plaintext. Padding bits can be seen as a safety guard to ensure the computation correctness. For instance, an addition of two messages may result in a carry, which is translated into one padding bit. In the same vein, by default the PBS uses one padding bit and is able to add many of them in it's output. In the `Crypto API`, homomorphic operations requiring padding bits are suffixed with `_with_padding`.

Automatic generation of the look-up table. The `Crypto API` provides an automatic generation for the look-up table associated to an arbitrary function f taking as input an `f64` and outputting an `f64`, mainly called in the PBS. It takes as inputs the encoding of the inputs, the encoding of the outputs, and the function f.

Benchmarks. Table 3 presents the time needed to compute a multiplication using two PBS's. The same setting as above is used. Table 4 gives the cost of a PBS and a key switch using the `Core API` and the `Crypto API` for the parameter set given in [2].

Table 3: Multiplication (in milliseconds) with two PBS's from `Crypto API` with $k = 1$, $B = 2^7$, $\ell = 3$ and 64-bit integers.

	$N = 1024$	$N = 2048$	$N = 4096$
$n = 630$	35.78	78.90	167.2
$n = 800$	45.82	100.1	213.3
$n = 1024$	58.77	127.8	273.6

Table 4: Comparison (in milliseconds) between `Core API` and `Crypto API` API. PBS: $n = 630$, $N = 1024$, $k = 1$, $B = 2^7$ and $\ell = 3$. LWE key switch: $q = 2^{32}$, $n_{\text{in}} = 1024$, $n_{\text{out}} = 630$, $B_{\text{KS}} = 4$ and $\ell_{\text{KS}} = 8$.

	CPU Core API		Crypto API
# bits	32	64	64
PBS	15.49	18.08	18.18
Key Switch	0.9912	2.292	2.262

3.3 Planned Demo

During the demo, we plan to quickly give an overview of `CONCRETE`, and then to run live tests for basic operations (both exact and approximate) and for neural network inference.

4 FUTURE WORK

The `CONCRETE` library offers an efficient and user-friendly interface to homomorphically compute over ciphertexts. There is a variety of ways to use encoding such as representing approximation of real numbers. Thanks to the programmable bootstrapping, it is possible to compute non-linear functions in addition to homomorphic additions and multiplications by scalars.

We plan to add other homomorphic operations and features in the next versions of the library. In parallel, we are implementing the GPU version of the `Core API` level and building an automatic API dealing with padding and noise automatically on its own. Future works also include some studies about enhancing the FFT performance and further support of SIMD operations. Finally, we want to work on the parallelization of the existing code.

REFERENCES

[1] Martin R. Albrecht, Rachel Player, and Sam Scott. 2015. On the Concrete Hardness of Learning with Errors. *Journal of Mathematical Cryptology* 9, 3 (2015), 169–203. https://bitbucket.org/malb/lwe-estimator/src/master/.

[2] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91. https://doi.org/10.1007/s00145-019-09319-x Earlier versions in ASIACRYPT 2016 and 2017.

[3] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology – EUROCRYPT 2015, Part I (Lecture Notes in Computer Science, Vol. 9056)*. Springer, 617–640. https://doi.org/10.1007/978-3-662-46800-5_24

[4] Matteo Frigo and Steven G. Johnson. 2005. The design and implementation of FFTW3. *Proceedings of the IEEE* 93, 2 (2005), 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation" http://www.fftw.org/.

[5] Craig Gentry. 2010. Computing arbitrary functions of encrypted data. *Communications of the ACM* 53, 3 (2010), 97–105. https://doi.org/10.1145/1666420.1666444 Earlier version in STOC 2009.

[6] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology – CRYPTO 2013, Part I (Lecture Notes in Computer Science, Vol. 8042)*. Springer, 75–92. https://doi.org/10.1007/978-3-642-40041-4_5

[7] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2013. On ideal lattices and learning with errors over rings. *Journal of the ACM* 60, 6 (2013), 43:1–43:35. https://doi.org/10.1145/2535925 Earlier version in EUROCRYPT 2010.

[8] Oded Regev. 2009. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM* 56, 6 (2009), 34:1–34:40. https://doi.org/10.1145/1568318.1568324 Earlier version in STOC 2005.

[9] Ronald L. Rivest, Len Adleman, and Michael L. Detouzos. 1978. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*. Academic Press, 165–179. Available at https://people.csail.mit.edu/rivest/pubs.html.

[10] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. 2009. Efficient public key encryption based on ideal lattices. In *Advances in Cryptology – ASIACRYPT 2009 (Lecture Notes in Computer Science, Vol. 5912)*. Springer, 617–635. https://doi.org/10.1007/978-3-642-10366-7_36