Polynomial Intermediate Representation for Fully Homomorphic Encryption

Last Updated: 2025-03-13

Contents

Introduction

As markets emerge, it's important to provide structure and standards that enable market development while avoiding choices that limit customer flexibility and agency. A typical structure in computation markets such as homomorphic encryption is a standard interface language that enables stratification of the market along natural boundaries of expertise. For example, standard application programming interfaces provide this kind of stratification between application source code - where product domain expertise is crucial - and "library" code - where algorithm and optimization expertise is the critical element. With such stratification in place, the different layers of expertise can work independently, so long as they agree to and abide by the standard interface between them.

Often, we coin common terms to refer to the participants in this kind of stratification boundary. One participant - often perceived as operating at a higher level of abstraction, is often termed the "caller", while the other - perceived as operating at a lower level - is often termed the "provider". We note that these boundaries between caller and provider often need to facilitate two-way communication: the provider must advertise to the caller what capabilities and limitations are available to the provider; and the provider must specify what operations to perform, with what parameter settings, and in what order.

This specification aims to define such a stratification boundary. Roughly speaking, the boundary we aim to define here lies between the homomorphic encryption libraries such as HEaaN, OpenFHE, and TFHE-rs on one hand, and purpose-built FHE acceleration hardware on the other hand. Specifically, we aim to define a standard intermediate representation (IR) of FHE code that is easy to generate by FHE libraries, and easy to translate to specific hardware instruction set architectures for FHE. This IR is a two-way communication channel, where hardware participants can advertise metadata such as supported parameter ranges and operations, and where library participants can specify parameter choices, relevant cryptographic assets (such as computation keys), and streams of instructions to process ciphertexts.

We leverage ideas in this specification from the emerging Zero Knowledge Proofs IR developed in part during the DARPA SIEVE program and further developed by ZKProofs.org: https://github.com/sieve-zk/ir/raw/main/v2.2.0/sieve-ir-v2.2.0.pdf

Polynomial IR

Mathematical Context

Mathematically, a ciphertext in a fully homomorphic encryption scheme based on the Ring Learning With Errors (RLWE) problem is a pair of elements – polynomials – of the quotient of the ring of integers of a cyclotomic field by a product of primes that splits completely. Concretely, a ciphertext can be represented as a pair of vectors, where each entry of each vector is a residue modulo the ciphertext modulus (the product of the primes that split completely), and the

length of each vector is the ring dimension (the degree of the polynomial defining the cyclotomic field).

Two other cipher representations are used in particular FHE schemes (the FHEW and CGGI – also called TFHE – schemes). The first one is based on the original Learning With Errors (LWE) problem, where ciphers may be seen as vectors of elements without any additional structure. The second one, called General Learning With Errors (GLWE) in the FHE community and Module Learning With Errors (MLWE) in the Post-Quantum community, is a generalisation of LWE and RLWE where a cipher is comprised of $k+1$ polynomials of degrees at most $N-1$, where k and N are two positive integers. The LWE and RLWE cases correspond to the extreme values $k = 0$ and $N = 0$. However, in this document we will primarily focus on RLWE ciphers since, to our knowledge, all GLWE operations may be expressed as combinations of RLWE operations. Unless stated otherwise, ciphers are thus assumed to be in RLWE form, and we will comment on where the LWE or GLWE case requires specific considerations.

Because the ciphertext modulus is typically much larger than a machine word and because the polynomial defining the cyclotomic field is of a very large degree, efficient implementations represent these vectors in what is known as *double-CRT representation*, based on the Chinese Remainder Theorem (CRT). The first CRT layer uses the residue number system (RNS) to decompose each residue polynomial modulo the ciphertext modulus into a vector of residue polynomials modulo each of the prime factors of the ciphertext modulus. This first layer produces the *coefficient representation* of the residue polynomial. Since each prime factor of the ciphertext modulus is typically chosen to fit into a single machine word, this form permits fast modular addition and subtraction of residue polynomials. The second CRT layer uses the negacyclic number theoretic transform (NTT) to convert a residue polynomial from the coefficient representation to the *evaluation representation*. While the evaluation representation also permits fast modular addition and subtraction of residue polynomials, it is the only representation that permits fast modular multiplication of residue polynomials. Thus, overall, the double-CRT representation allows us to represent each vector of a ciphertext as a matrix of small residues that enables efficient arithmetic by component-wise modular operations in machine words.

Here again the CGGI scheme is an outlier as allowing for a much smaller modulus, which may also be a power of 2. In this case, it may be more efficient to use Fourier Transforms instead of NTTs. Most of the description below still applies to this case (with a number of moduli set to 1), and we will comment explicitly on where the specificities of the CGGI scheme require separate considerations.

Organizational Preface

In the sections that follow, we specify the basic types and baseline instructions that any hardware compliant with this IR are required to implement. We also lay out some examples of optional data types and instructions that some hardware may choose to implement. In between these two categories, we define the notion of a *gadget* as an instruction that is implementable in

terms of basic type and baseline operations and, if available, optional types and optional operations, that some hardware may choose to implement directly.

Basic Data Types

These data types are required to be implemented by any hardware compliant with this specification.

Polynomial

The first basic data type is a polynomial. The data required to specify a polynomial is its vector of components (coefficients for the coefficient representation, or values for the evaluation representation), and the following metadata fields:

- A field specifying whether it is "integer" or “non-integer”. The “integer” case is intended to support components modulo prime and composite integer moduli, and the “non-integer” case is intended to support components that are fixed point (64-bit integer with binary point location) and floating point values (e.g. float32).
- A field specifying the “ring dimension”. This field is included to support ring switching.

Scalar

The second basic data type is a scalar. The data required to specify a scalar is its value and the following metadata fields:

- A field specifying whether it is "integer" or “non-integer”. The “integer” case is intended to support values modulo prime and composite integer moduli, and the “non-integer” case is intended to support fixed point (64-bit integer with binary point location) and floating point values (e.g. float32).

Baseline Instructions

These operations are required to be implemented by any hardware compliant with this specification, so long as the upward advertisements by the hardware support the operations specified.

For the following instructions, we shall use the following notation:

- $a = (a_0, a_1, \ldots, a_{N-1}), b = (b_0, b_1, \ldots, b_{N-1})$ are input polynomials.
- s is an input scalar.
- q is a modulus.
- $f = (f_0, f_1, \ldots, f_{N-1})$ is an output polynomial.
- $[k]_m$ is short for $(k \bmod m)$.

Polynomial Addition

Takes as input two polynomials, *a* and *b*, and a modulus, *q*, and returns a single polynomial modulo *q*. The return value is the component-wise sum of *a* and *b* reduced modulo *q*.

$$f = sr\_addp(a, b, q) \qquad \rightarrow f_i = [a_i + b_i]_q$$

Polynomial Scalar Addition

Takes as input a scalar, *s*, a polynomial, *a*, and a modulus, *q*, and returns a single polynomial modulo *q*. For *a* in the evaluation-representation, the operation adds *s* to every component of *a*, and returns the component-wise sum reduced modulo *q*. For *a* in the coefficient-representation, the operation adds *s* to the first coefficient of *a* modulo *q* and returns the resulting polynomial.

$$f = sr\_addps(a, s, q) \qquad \rightarrow f_i = [a_i + s]_q$$

$$f = sr\_addps\_coeff(a, s, q) \qquad \rightarrow f_0 = [a_0 + s]_q, f_{i>0} = [a_{i>0}]_q$$

Polynomial Negation

Takes as input a single polynomial, *a*, and a modulus, *q*, and returns a single polynomial. The return value is the component-wise negation of each component of *a* modulo *q*. This operation is explicitly included to allow polynomial subtraction to be defined as negation followed by addition.

$$f = sr\_negp(a, q) \qquad \rightarrow f_i = [-a_i]_q$$

Polynomial Subtraction

Takes as input two polynomials, *a* and *b*, and a modulus, *q*, and returns a single polynomial modulo *q*. The return value is the component-wise difference of the first input polynomial, *a*, and the second input polynomial, *b*, reduced modulo *q*.

$$f = sr\_subp(a, b, q) \qquad \rightarrow f_i = [a_i - b_i]_q$$

Polynomial Scalar Subtraction

Takes as input a scalar, *s*, a polynomial, *a*, and a modulus, *q*, and returns a single polynomial modulo *q*. For *a* in the evaluation-representation, the operation subtracts *s* from every component of *a*, and returns the component-wise difference reduced modulo *q*. For *a* in the coefficient-representation, the operation subtracts *s* from the first coefficient of *a* modulo *q* and returns the resulting polynomial.

$$f = sr\_subps(a, s, q) \qquad \rightarrow f_i = [a_i - s]_q$$

$$f = sr\_subps\_coeff(a, s, q) \qquad \rightarrow f_0 = [a_0 - s]_q, f_{i>0} = [a_{i>0}]_q$$

Polynomial Multiplication

Takes as input two polynomials, *a* and *b*, and a modulus, *q*, and returns a single polynomial modulo *q*. The return value is the component-wise product of *a* and *b* reduced modulo *q*. Note that this only represents the polynomial product of *a* and *b* if they are both in the evaluation representation.

$$f = sr\_mulp(a, b, q) \qquad \rightarrow f_i = [a_i \cdot b_i]_q$$

Polynomial Scalar Multiplication

Takes as input a scalar, *s*, a polynomial, *a*, and a modulus, *q*, and returns a single polynomial modulo *q*. The return value is the component-wise product of every component of *a* by *s* reduced modulo *q*.

$$f = sr\_mulps(a, s, q) \qquad \rightarrow f_i = [a_i \cdot s]_q$$

Negacyclic NTT

Takes as input a polynomial, *a*, and a modulus, *q*, and returns the polynomial that is the result of applying the negacyclic number theoretic transformation (NTT) relative to *q* to *a*. More precisely, this takes *a* from the coefficient representation (time domain) relative to *q* to the evaluation representation (frequency domain) relative to *q*. The action of this instruction is represented by the following formula:

$$f = sr\_NTT(a, q) \qquad \rightarrow f_i = \left[\sum_{j=0}^{N-1} a_j \cdot \psi^{j+2ji}\right]_q$$

where ψ denotes a 2*N*-th primitive root of unity modulo *q*.

Inverse Negacyclic NTT

Takes as input a polynomial, *a*, and a modulus, *q*, and returns the polynomial that is the result of applying the inverse negacyclic number theoretic transformation (INTT) relative to *q* to *a*. More precisely, this takes *a* from the evaluation representation (frequency domain) relative to *q* to the coefficient representation (time domain) relative to *q*. The action of this instruction is represented by the following formula:

$$f = sr\_iNTT(a, q) \qquad \rightarrow f_i = \left[\psi^{-i} N^{-1} \sum_{j=0}^{N-1} a_j \cdot \psi^{-2ji}\right]_q$$

where ψ denotes a 2*N*-th primitive root of unity modulo *q*.

Automorphism

Takes as input a single polynomial, *a*, and an odd integer between 1 and 2*N* - 1, where *N* is the ring dimension of the polynomial. Returns the result of applying the Galois automorphism corresponding to the given odd integer to *a*. There are separate operations for evaluation representation and coefficient representation, the latter of which additionally requires a modulus, *q*, as a parameter.

$$f = sr\_automorph\_eval(a, k) \qquad \rightarrow f_i = a_{\frac{[k(2i+1)-1]_{2N}}{2}}$$

$$f = sr\_automorph\_coeff(a, k, q) \qquad \rightarrow f_{[ki]_N} = \begin{cases} a_i & [ki]_{2N} < N \\ [-a_i]_q & [ki]_{2N} \geq N \end{cases}$$

Halt

This operation tells the machine to stop and notifies the host. The mechanism of notification is machine-specific.

Gadgets

These are operations that are implementable using the basic types and baseline operations or, potentially, the optional operations, but which can optionally be supported by direct implementations in hardware.

One defines a gadget by defining a macro and giving it a name. The compiler can replace the named macro with the appropriate code. Some examples of potential gadgets are things like modulus switching and keyswitching.

Example: Compound Types

Multi-Residue Polynomial (MRP)

This is a set of polynomials together with a set of moduli where the two sets are required to be the same size.

This is a convenience type for explicitly assigning a modulus to each polynomial. The set of moduli can be accessed as an index set via the subfield “.base”.

Multi-Residue Scalar (MRS)

This is a set of scalars together with a set of moduli where the two sets are required to be the same size.

This is a convenience type for explicitly assigning a modulus to each polynomial. The set of moduli can be accessed as an index set via the subfield ".base".

Example: Multi-Residue Polynomial Operations, Fast Basis Conversion, and Rescaling

Note that these gadgets use the compound type gadgets of multi-residue polynomials and the optional multi-residue scalars defined above. Note also that these gadgets assume that both operands have the same base.

```
def mr_addp (x: MRP, y: MRP):
      z = MRP(x.base)
      for q in x.base:
            z[q]=sr_addp(x[q], y[q], q)
      return z
```

```
def mr_subp (x: MRP, y: MRP):
      z = MRP(x.base)
      for q in x.base:
            z[q]=sr_subp(x[q], y[q], q)
      return z
```

```
def mr_mulp (x: MRP, y: MRP):
      z = MRP(x.base)
      for q in x.base:
            z[q]=sr_mulp(x[q], y[q], q)
      return z
```

```
def mr_mulps (x: MRP, s: MRS):
      z = MRP(x.base)
      for q in x.base:
            z[q]=sr_mulps(x[q], s[q], q)
      return z
```

```
def mr_addps (x: MRP, s: MRS):
      z = MRP(x.base)
      for q in x.base:
            z[q]=sr_addps(x[q], s[q], q)
      return z
```

```
def mr_ntt (x: MRP):
      z = MRP(x.base)
      for q in x.base:
            z[q]=sr_ntt(x[q], q)
      return z
```

```
def mr_intt (x: MRP):
      z = MRP(x.base)
      for q in x.base:
            z[q]=sr_intt(x[q], q)
      return z
```

The above gadgets mainly offer convenience.

The following gadgets can offer significant information as to the implementation on a given architecture:

$\boldsymbol{z = FastBaseConvert(x, B_{tar})}$

- Input: $x = \{x_0, x_1, \dots, x_l\}$, a coefficient-mode MRP in base $B_{src} = \{q_0, q_1, \dots, q_l\}$
- Input: $B_{tar} = \{p_0, p_1, \dots, p_k\}$, a base, describing the target moduli
- Output: $z = \{z_0, z_1, \dots, z_k\}$, a coefficient-mode MRP in base $B_{tar} = \{p_0, p_1, \dots, p_k\}$

Mathematical Description:

$$z_i = [z]_{p_i} = \sum_{q_j \in B_{src}} \left[[x_j \cdot \hat{q}_j]_{q_j} \cdot q^*_{j,i} \right]_{p_i}$$

With $\hat{q}_j$ and $q^*_{j,i}$ being scalars.

Pseudo-Code:

```
def FastBaseConvert (x: MRP, target_base: set):
      z = MRP(target_base)

      for q in x.base:
            x[q]=sr_mulps(x[q], q_hat(source_base, q), q)

      for p in target_base:
            for q in x.base:
                  temp = sr_mulps(x[q], q_star(q,p), p)
                  z[p] = sr_addp(z[p], temp, p)
      return z
```

$\boldsymbol{z = RescaleFBC(x, B_{resccale})}$

- Input: $x = \{x_0, x_1, \dots, x_l\}$, a coefficient-mode MRP in base $B_{src} = \{q_0, q_1, \dots, q_l\}$
- Input: $B_{resccale} = \{p_0, p_1 \dots, p_k\} \subset B_{src}$
- Output: $z = \{z_0, z_1, \dots, z_m\}$, a coefficient-mode MRP in base $B_{tar} = B_{src} \backslash B_{resccale}$

Mathematical Description:

$$y = FastBaseConvert(x, B_{src} \backslash B_{resccale})$$
$$z_i = [z]_{q_i \in B_{tar}} = [(x_i - y_i) \cdot \tilde{q}_i]_{q_i}$$

With $\tilde{q}_i$ being scalars.

Pseudo-Code:

```
def RescaleFBC (x: MRP, rescale_base: set):
     target_base = x.base - rescale_base        # set difference
     z = MRP(target_base)
     y = FastBaseConvert (x, target_base)

     for q in target_base:
          temp = sr_subp(x[q], y[q], q)
          z[q] = sr_mulp(temp, q_wave(x.base, q), q)
     return z
```

Example: The GSW/RLWE External Product

Note that the following gadget requires the optional instructions required for TFHE and FHEW.

The external product between GSW and RLWE ciphers is the main ingredient of the TFHE scheme, used as the main building-block for the two main bootstrapping algorithms (circuit and programmable). Here we provide a high-level description to show how it can be decomposed into lower-level operations, as an example of a gadget that may be implemented in hardware.

Note: The choice of GSW/RLWE External Product as gadget is somewhat arbitrary, and presented for illustration purpose only. Concrete hardware implementations may use higher-level gadgets (such as Blind Rotation) to maximise the opportunities for dataflow optimisation or lower-level ones (such as the Half external product) to provide compilers with more flexibility.

Parameters:

- l: Number of levels for Gadget Decomposition
- B: Decomposition basis for Gadget Decomposition
- F: A negacyclic-Fourier-transform-like function; it could be a negacyclic Fourier transform or negacyclic NTT
- iF: Inverse of the function F

Inputs:

- gsw: A GSW cipher comprising 4 *l* polynomials with integer coefficients.
- rlwe_in: An RLWE cipher comprising 2 polynomials with integer, real, or complex coefficients

Output: an RLWE cipher

High-level description:

1. Run the Gadget Decomposition with parameters *(l,B)* on each of the two polynomials of rlwe_in, producing 2 arrays a1, a2 of l polynomials.
2. Generate the polynomials p11, p12, p21, and p22 as follows:
    a. p11 is the dot product of the first l polynomials of gsw with a1
    b. p12 is the dot product of the polynomials l to 2l-1 of gsw with a1

    c. p21 is the dot product of the polynomials 2*l* to 3*l-1* of gsw with a2
    d. p22 is the dot product of the polynomials 3*l* to 4*l-1* of gsw with a2
3. Return the RLWE cipher (p11+p21, p12+p22).

Optional Operations

These are operations that are not implementable using the baseline operations, but might be available in some hardware. Similarly, these are data types that might be available in some hardware.

One potential example of an optional operation is one that would generate a random modular polynomial. Random modular polynomials get used a fair bit in keyswitching.

This also provides an avenue for scheme-specific operations.

Negacyclic Fourier Transform

This operation is a possible alternative to the Negacyclic NTT for hardware supporting only the FHEW and/or TFHE schemes. It takes as input a polynomial with complex coefficients and returns another polynomial with complex coefficients obtained as follows, where *N* denotes the degree of the cyclotomic polynomial (we assume *N* is a power of 2):

- Starting from the input polynomial p_1, define the polynomial p_2 whose coefficients are those of p_1 multiplied by powers of a (2N)th primitive complex root of unity, i.e. a complex number of the form exp(i π r / N) where r is a fixed odd integer. If p_1 is written as
  $$p_1 = p_{1,0} + p_{1,1} X + p_{1,2} X^2 + \ldots + p_{1,N-1} X^{N-1},$$
  the polynomial p_2 may thus be written as
  $$p_2 = p_{1,0} + \exp(i \pi r / N)\, p_{1,1} X + \exp(2 i \pi r / N)\, p_{1,2} X^2 + \ldots + \exp((N-1) i \pi r / N)\, p_{1,N-1} X^{N-1},$$
- Define the polynomial p_3 whose coefficients form the Fourier transform of those of p_2 .
- Output p_3 .

Note that the above sequence of operations is illustrative only. Actual implementations may organize the computation differently provided the relation between input and output is kept.

The input of the Fourier transform may, depending on the implementation, be represented in integer, fixed-point, or floating-point form. For implementations where the input is in integer form, the negacyclic Fourier transform must include a conversion step from integer to fixed- or floating-point representation.

Inverse Negacyclic Fourier Transform

This operation is a possible alternative to the Inverse Negacyclic NTT for hardware supporting only the FHEW and/or TFHE schemes. It must be used if and only if the Negacyclic Fourier Transform is used in place of the Negacyclic NTT. It may be computed as follows:

- Starting from the input polynomial p_1, define the polynomial p_2 whose coefficients form the inverse Fourier transform of those of p_1 .

- Define the polynomial p_3 whose coefficients are those of p_2 multiplied by powers of the inverse of the root of unity used for the Negacyclic Fourier transform. If p_2 is written as
  $$p_2 = p_{2,0} + p_{2,1} X + p_{2,2} X^2 + \ldots + p_{2,N-1} X^{N-1},$$
  the polynomial p_3 may thus be written as
  $$p_3 = p_{2,0} + \exp(-i \pi r / N)\, p_{2,1} X + \exp(-2 i \pi r / N)\, p_{2,2} X^2 + \ldots + \exp(-(N-1) i \pi r / N)\, p_{2,N-1} X^{N-1}.$$
- Output p_3 .

The output of the inverse Fourier transform may, depending on the implementation, be represented in integer, fixed-point, or floating-point form. For implementations where the output is in integer form, the negacyclic Fourier transform must include a conversion step from fixed- or floating-point representation to integer.

Negacyclic Rotation

This operation is only required for the TFHE scheme. It takes a polynomial p_1 and an integer k as input and outputs another polynomial p_2 with coefficients given by:

$$p_{2,i} = (-1)^{\lfloor (i+k) / N \rfloor}\, p_{2,i+k},$$

Evaluated modulo the (prime or power-of-2) modulus.

Coefficient extraction

This operation is only required for the TFHE and FHEW schemes. It takes as input a polynomial p and an integer index i, required to be smaller than the length of the polynomial, and returns a scalar value equal to the ith component of p.

Gadget Decomposition

This operation is only required for the TFHE and FHEW schemes. It takes as inputs a polynomial p_0 and two positive integers l (number of decomposition levels) and b (decomposition basis), and returns an array of l polynomials $p_1, p_2, \ldots, p_l$ such that

- Each coefficient of each input polynomial is in the range [0, b-1] if using unsigned integers or [-(b+1)/2, b/2] if using signed integers.
- The coefficients of $(p_1 + b\, p_2 + \ldots + b^{l-1}\, p_l)\, K$ differ from those of p_0 by less than E in absolute values, where K and E are parameters possibly dependent on l and b.

Hardware for TFHE may be said conforming provided it supports Gadget Decomposition for at least one value of (l, b, K, E) compatible with the other supported parameters.

Parameter Context Specification

A block containing the following parameters may appear more than one time in an IR instance and immediately precedes the sequence of instructions to which these parameters apply. The ability to include more than one parameter specification block is intended to support scheme-switching.

FHE Scheme

E.g. CKKS, BGV, BFV, TFHE, FHEW.

Ring Dimension

Required to be a power of two.

Basic Arithmetic Field Size

E.g. 32 bits or 64 bits.

Length of the Ciphertext Modulus Chain

Ciphertext Modulus

Number of decomposition levels for keyswitching

This is important because it determines the amount of memory required for keyswitching.

Parameters specific to the TFHE and/or FHEW schemes (Optional)

The following parameters are only relevant for hardware targeting the TFHE and/or FHEW schemes. These are only valid in a block that specifies the scheme as “TFHE” or “FHEW”.

Number of decomposition levels for Gadget Decomposition

Decomposition basis for Gadget Decomposition

Decomposition basis for key-switching

Modulus

Information that should flow upward

A hardware solution wants to advertise upward what it can do to the compiler. What parameters are supported? What parameters are optimal?

The following information should flow upward from the IR to the compiler and software layers:

- Ring dimension
- Field size

- Length of modulus chain
- Gadgets
- Optional operations

Information that should flow downward

This should include the parameter choices and the instructions used by the program.

The following information should flow downward from the IR to the assembler and hardware layers:

- Ring dimension
- List of primes for RNS representation
- Coefficient form or point-value form
- Precision of coefficients
- Prime congruence
- Instruction sequence

Future enhancements: Cost modeling and optimization

This is intended to build on the above advertisements of information upward, providing further avenues of optimization. For example, the hardware could advertise the cost of individual operations and the available memory.

Appendix: Syntax (Under Development)

The current in-progress draft of the syntax is here.

Examples (Under Development)

The TFHE Programmable Bootstrapping (PBS)

Parameters (example value):

```
LWE_WORD_SIZE (12 bits)
LWE_DIMENSION (938)
RLWE_WORD_SIZE (32 bits)
RLWE_DIMENSION (2048)
DECOMPOSITION_LEVELS (3)
LOG_DECOMPOSITION_BASIS (6)
MODULUS (1 << 32)
```

High-level pseudocode (C-like syntax):
*NOTE: This pseudo-code is only provided as an example to illustrate which operations may be required in a TFHE workflow. It does **not** constitute a reference description of the PBS nor specifies how the PBS should be implemented.*

```
void pbs(const lwe_coeff_t* const lwe_location,
         const rlwe_coeff_t* const lut_location,
         const rlwe_coeff_t* const pbs_key_location,
         const rlwe_coeff_t* rlwe_temp,
         const rlwe_coeff_t* gsw_temp,
         const lwe_coeff_t* lwe_output)
{
  // Cleartext negacyclic rotation
  load(lut_location, rlwe_temp, 2 * RLWE_DIMENSION);
  negacyclic_rotation(rlwe_temp, rlwe_temp, lwe_location[0],
                      RLWE_DIMENSION, MODULUS);

  // Blind rotation
  for (unsigned int i = 1; i <= LWE_DIMENSION; ++i) {
    negacyclic_rotation(rlwe_temp, rlwe_temp + 2 * RLWE_DIMENSION,
                        lwe_location[i], RLWE_DIMENSION, MODULUS);
    mod_sub(rlwe_temp + 2 * RLWE_DIMENSION, rlwe_temp,
            2 * RLWE_DIMENSION,
            rlwe_temp + 4 * RLWE_DIMENSION, MODULUS);
    gadget_decomposition(rlwe_temp + 4 * RLWE_DIMENSION,
                         rlwe_temp + 6 * RLWE_DIMENSION,
                         DECOMPOSITION_LEVELS,
                         LOG_DECOMPOSITION_BASIS, MODULUS);
    zero(rlwe_temp + 2 * RLWE_DIMENSION, 2 * RLWE_DIMENSION);

    for (unsigned int j = 0; j < DECOMPOSITION_LEVELS; ++j) {
      forward_transform(rlwe_temp + (6 + 2*j) * RLWE_DIMENSION,
                        RLWE_DIMENSION, MODULUS);
      forward_transform(rlwe_temp + (6 + 2*j + 1) * RLWE_DIMENSION,
                        RLWE_DIMENSION, MODULUS);
      load(pbs_key_location
             + 4 * RLWE_DIMENSION * (i * DECOMPOSITION_LEVELS + j),
           gsw_temp, 2 * RLWE_DIMENSION);
      mod_mul(rlwe_temp + (6 + 2*j) * RLWE_DIMENSION, gsw_temp,
              gsw_temp, 2 * RLWE_DIMENSION, MODULUS);
      mod_add(rlwe_temp + 2 * RLWE_DIMENSION, gsw_temp,
              rlwe_temp + 2 * RLWE_DIMENSION,
```

```
            2 * RLWE_DIMENSION, MODULUS);
      load(pbs_key_location
             + 4 * RLWE_DIMENSION * (i * DECOMPOSITION_LEVELS + j)
             + 2 * RLWE_DIMENSION,
           gsw_temp, 2 * RLWE_DIMENSION);
      mod_mul(rlwe_temp + (6 + 2*j) * RLWE_DIMENSION, gsw_temp,
              gsw_temp, 2 * RLWE_DIMENSION, MODULUS);
      mod_add(rlwe_temp + 2 * RLWE_DIMENSION,
              gsw_temp + RLWE_DIMENSION,
              rlwe_temp + 2 * RLWE_DIMENSION,
              RLWE_DIMENSION, MODULUS);
      mod_add(rlwe_temp + 3 * RLWE_DIMENSION,
              gsw_temp, rlwe_temp + 3 * RLWE_DIMENSION,
              RLWE_DIMENSION, MODULUS);
    }
    inverse_transform(rlwe_temp + 2 * RLWE_DIMENSION,
                      RLWE_DIMENSION, MODULUS);
    inverse_transform(rlwe_temp + 3 * RLWE_DIMENSION,
                      RLWE_DIMENSION, MODULUS);
    mod_add(rlwe_temp, rlwe_temp + 2 * RLWE_DIMENSION, rlwe_temp,
            2 * RLWE_DIMENSION, MODULUS);
  }

  // Sample extraction (conversion from RLWE to LWE)
  lwe_output[0] = rlwe_temp[RLWE_DIMENSION];
  for (unsigned int i = 1; i <= LWE_DIMENSION; ++i) {
    lwe_output[i] =
      mod_negative(rlwe_temp[(RLWE_DIMENSION - i) % RLWE_DIMENSION],
                   MODULUS);
  }
}
```

High-level pseudocode (C++-like syntax with Polynomial template):
*NOTE: This pseudo-code is only provided as an example to illustrate which operations may be required in a TFHE workflow. It does **not** constitute a reference description of the PBS nor specifies how the PBS should be implemented.*

```
typedef Polynomial<rlwe_coeff_t,RLWE_DIMENSION,MODULUS> poly_t;

void pbs(const lwe_coeff_t* const lwe_location,
         const poly_t& lut,
```

```
        const poly_t* const pbs_key,
        const poly_t* rlwe_temp,
        const poly_t* gsw_temp,
        const lwe_coeff_t* lwe_output)
{
  // Cleartext negacyclic rotation
  load(lut, rlwe_temp, 2); // Question: replace with a hint?
  negacyclic_rotation(rlwe_temp, rlwe_temp, lwe_location[0]);

  // Blind rotation
  for (unsigned int i = 1; i <= LWE_DIMENSION; ++i) {
    negacyclic_rotation(rlwe_temp, rlwe_temp + 2, lwe_location[i]);
    mod_sub(rlwe_temp + 2, rlwe_temp, rlwe_temp + 4, 2);
    gadget_decomposition(rlwe_temp + 4, rlwe_temp + 6,
                         DECOMPOSITION_LEVELS,
                         LOG_DECOMPOSITION_BASIS);
    zero(rlwe_temp + 2, 2);

    for (unsigned int j = 0; j < DECOMPOSITION_LEVELS; ++j) {
      forward_transform(rlwe_temp + 6 + 2*j);
      forward_transform(rlwe_temp + 6 + 2*j + 1);
      load(pbs_key_location + 4 * (i * DECOMPOSITION_LEVELS + j),
           gsw_temp, 2);
      mod_mul(rlwe_temp + 6 + 2*j, gsw_temp, gsw_temp, 2);
      mod_add(rlwe_temp + 2, gsw_temp, rlwe_temp + 2, 2);
      load(pbs_key_location
             + 4 * (i * DECOMPOSITION_LEVELS + j)
             + 2, gsw_temp, 2);
      mod_mul(rlwe_temp + 6 + 2*j, gsw_temp, gsw_temp, 2);
      mod_add(rlwe_temp + 2, gsw_temp + 1, rlwe_temp + 2);
      mod_add(rlwe_temp + 3, gsw_temp, rlwe_temp + 3);
    }
    inverse_transform(rlwe_temp + 2);
    inverse_transform(rlwe_temp + 3);
    mod_add(rlwe_temp, rlwe_temp + 2, rlwe_temp, 2);
  }

  // Sample extraction (conversion from RLWE to LWE)
  lwe_output[0] = get_coefficient(rlwe_temp[1], 0);
  for (unsigned int i = 1; i <= LWE_DIMENSION; ++i) {
    lwe_output[i] =
      mod_negative(get_coefficient(
```

```
                    rlwe_temp[0],
                    (RLWE_DIMENSION - i) % RLWE_DIMENSION),
                  MODULUS);
  }
}
```