Polynomial Intermediate Representation for Fully Homomorphic Encryption

Last Updated: 2025-05-19

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 lower-level 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 fully homomorphic encryption (FHE) libraries like HEaaN, OpenFHE, and TFHE-rs on one hand, and purpose-built FHE acceleration hardware on the other. Specifically, we aim to define a standard intermediate representation (IR) of FHE code that is both easily generated by FHE libraries, and also easily translated 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 IR for zero-knowledge proofs 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 distinct primes. 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), and the length of each vector is the ring dimension (the degree of the polynomial defining the cyclotomic field).

It is worth mentioning that two other ciphertext representations are used in some FHE schemes -- in particular, the FHEW and TFHE (also called CGGI) schemes. The first one is based on the original learning with errors (LWE) problem, where ciphertexts may be seen as vectors of elements without any additional structure. The second one, known as general learning with errors (GLWE) or module learning with errors (MLWE) , is a generalisation of LWE and RLWE where a ciphertext 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 $N = 1$ and $k = 1$, respectively. However, in this document we will primarily focus on RLWE ciphers because, to our knowledge, most GLWE operations may be expressed as combinations of RLWE operations. As such, unless stated otherwise, ciphertexts in this document are thus assumed to be RLWE ciphertexts, 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 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 TFHE scheme is an outlier, as it allows 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 (FFTs) 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 TFHE 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 specification are required to implement, subject to any advertised limits on parameters (e.g. available ring dimensions or moduli) supported by the hardware.

We then define the notion of *gadget* as an instruction that is implementable in terms of basic type and baseline operations that some hardware may choose to implement directly.

Next, we lay out some examples of *optional data types* and *optional instructions* that some hardware may choose to implement. Within the space of optional instructions, we recognize at least two initial likely points of focus: first, providing support for non-integer versions of the baseline operations, and second, much more broadly, to provide direct hardware implementations of higher-level functionality.

Finally, we define *upward advertisements*: capabilities expressible in this IR that can be advertised by hardware upward to the software layer, allowing compilers to take advantage of those capabilities.

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 the polynomial 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 (e.g. 64-bit integer with binary point location) or floating point values (e.g. float32).
- A field specifying the “ring dimension”.

In what follows, we will often refer to polynomials according to this first metadata field – i.e. integer polynomials and non-integer polynomials. Additionally, based on context, we sometimes refer to integer polynomials as single-residue polynomials for clarity.

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 the scalar 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 (e.g. 64-bit integer with binary point location) or floating point values (e.g. float32).

In what follows, we will often refer to scalars according to this metadata field – i.e. integer scalars and non-integer scalars.

Baseline Instructions

These operations are required to be implemented by any hardware compliant with this specification, under the limitations (e.g. available ring dimensions or moduli) advertised upward by the hardware.

For the following instructions, we shall use this notation:

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

Polynomial Addition

Takes as input two integer polynomials, *a* and *b*, and a modulus, *q*, and returns a single integer 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 an integer scalar, *s*, an integer polynomial, *a*, and a modulus, *q*, and returns a single integer 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 integer polynomial, *a*, and a modulus, *q*, and returns a single integer 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 integer polynomials, *a* and *b*, and a modulus, *q*, and returns a single integer 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 an integer scalar, *s*, an integer polynomial, *a*, and a modulus, *q*, and returns a single integer 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 integer polynomials, *a* and *b*, and a modulus, *q*, and returns a single integer 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 an integer scalar, *s*, an integer polynomial, *a*, and a modulus, *q*, and returns a single integer 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 an integer polynomial, *a*, and a modulus, *q*, and returns the integer 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 relative to *q* to the evaluation representation 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*. Hardware implementations may either support a way to set the value of ψ to be used for each modulus, and/or a mechanism (e.g. via a look-up table) to set it automatically.

Inverse Negacyclic NTT

Takes as input an integer polynomial, *a*, and a modulus, *q*, and returns the integer 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*. Hardware implementations may either support a way to set the value of ψ to be used for each modulus, and/or a mechanism (e.g. via a look-up table) to set it automatically.

General Permutation

Takes as input a single polynomial, *a*, and two parameters vectors, *srcs* and *signs*, of the same length as *a*, where the entries of *srcs* are the integers between 0 and N-1 (representing the indices of *a* under the desired permutation), and the entries of *signs* are all either 1 or -1 (representing whether to flip the sign of the corresponding entry). Returns a polynomial of values originating from ***a*** according to indices from *srcs*, with sign-flips according to vector *signs.* Implementing the sign flip requires an additional parameter *q,* the modulus.

$$f = sr\_permute(a, srcs, signs, q) \qquad \rightarrow f_i = \begin{cases} a_{srcs_i} & signs_i = 1 \\ [-a_{srcs_i}]_q & signs_i = -1 \end{cases}$$

Halt

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

Gadgets

This specification recognizes that relying solely on baseline, single-residue polynomial level operations and data types may hinder the development of higher-level optimizations directly in hardware. This limitation can also result in large, cumbersome code with many repetitions.

To address these issues, this specification introduces the notion of *gadget*. A *gadget* (respectively, *data gadget*) is a compound operation (respectively, data type) that can be expressed completely in terms of baseline operations (respectively, basic types). The purpose of defining the notion of gadget is to allow hardware vendors to provide direct hardware implementations of higher-level operations, while maintaining the ability for any hardware implementing this standard to be able to run that higher-level operation by using its definition in terms of baseline operations. Gadgets can then be incorporated into the code used by everyone, but can be processed by the vendor's backend compiler to run directly if the hardware supports a gadget directly.

Due to the enormous span of possible gadgets of widely varying levels of complexity, we propose the following initial mutually exclusive breakdown into classes:

- Class I (Simple): The proposed implementation of a gadget in this class in terms of baseline/optional operations must be seen to be exactly equal by inspection to the semantics of the gadget as described.
- Class II (Complex): The proposed implementation of a gadget in this class in terms of baseline/optional operations must be shown (say, using formal methods) to be equivalent to the semantics of the gadget as described. Proof of this formal equivalence must be provided when submitting such a gadget to be included in this specification.

Gadget Approval Process

The proposed process of adding a new gadget definition to this specification shall include, at minimum, a presentation of the proposed gadget to the FHETCH technical committee, who will decide according to criteria such as sufficient verification data, usability, generalization versus existing operation and gadgets, and more.

Example Potential Gadgets

Note that the proposed implementations of Class I gadgets in the current version of this specification may contain pseudocode operators that are not defined by or included in this specification. These operators are used for mathematical rigor (such as constructors of data types with empty, zero, or other predefined values) and readability purposes (such as loops, and general index arithmetic).

Polynomial Gadgets

These gadgets work with the basic polynomial data type.

For convenience when writing pseudocode here, we define syntax for a zero polynomial constructor, which takes as input a ring dimension N, and returns a polynomial of ring dimension N, and whose coefficients are all equal to zero.

$$f = sr\_zeros(N) \qquad \rightarrow f_i = 0$$

Galois Automorphism

Takes as input a single integer 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.

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

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

Pseudocode using the general permutation baseline operation:

```
def sr_automorph_eval (x: SRP, k: int):
     N = x.ring_dimension
     srcs = zeros(N)          # initialize an array
     signs = zeros(N)         # initialize an array
     for i in range(N):
          srcs[i] = ((k * (2 * i + 1) - 1) % 2*N) // 2
          signs[i] = 1
     return sr_permute(x, srcs, signs, q=DontCare)
```

```
def sr_automorph_coeff (x: MRP, k: int, q: int):
     N = x.ring_dimension
     srcs = zeros(N)          # initialize an array
     signs = zeros(N)         # initialize an array
     for i in range(N):
          srcs[(k * i) % N] = i
          signs[(k * i) % N] = (-1) ** ((k * i) % (2 * N)) // N
     return sr_permute(x, srcs, signs, q)
```

Notes:

1. Operators +,-,*,%,//,** in the macro have their python-syntax meaning (i.e. addition, subtraction, multiplication, modulo, integer division, power).

2. In these two examples, none of the variables/inputs `k,q, N, srcs, signs, i` are of datatypes this specification provides operations for. Only `x` and the return value are of data type (single residue polynomial) that has operators specified for.

Negacyclic Rotation Automorphism

Takes as input a single coefficient mode integer polynomial, *a*, and a parameter *offset* (in range 0 and N-1), and a modulus *q,* where *N* is the ring dimension of the polynomial. Returns the result of applying cyclic rotation of the polynomials' coefficients by the given *offset*, including accounting for required sign changes:

$$f = sr\_rot\_automorph\_coeff(a, offset, q)$$

$$\rightarrow f_i = \begin{cases} a_{[i+offset]_N} & i + offset < N \\ \left[-a_{[i+offset]_N}\right]_q & i + offset \geq N \end{cases}$$

```
def sr_rot_automorph_coeff (x: MRP, offset: int, q: int):
     N = x.ring_dimension
     srcs = zeros(N)          # initialize an array
     signs = zeros(N)         # initialize an array
     for i in range(N):
          srcs[i] = (i + offset) % N
          signs[i] = (-1) ** ((i + offset) // N)
     return sr_permute(x, srcs, signs, q)
```

Single-Residue Polynomial Array (SRPA)

This is a data type representing an array of single-residue polynomials (SRPs). We provide syntax for three constructors for pseudocode:

- `SRPA()` creates an empty SRPA
- `SRPA(n: uint)` creates an SRPA containing n SRPs (initialised in an implementation-dependent way)
- `SRPA(p1, p2, …, pn)` creates an SRPA containing the SRPs p1, p2, …, pn

If a is an SRPA, u an unsigned integer, and p an SRP,

- `a.length` returns the number of elements in a
- `a[u]` returns the element u in a if `u < a.length`
- `a[u] = p` sets the element u of a to p if `u < a.length`

Batch Fourier Transforms

These are direct and inverse Fourier transforms acting on arrays of SRPs.

Pseudocode:

```
def sr_batch_ft (x: SRPA):
    y = SRPA(x.length)
    for i in range(x.length):
        y[i] = sr_ft(x[i])
    return y
```

Pseudocode:

```
def sr_batch_ift (x: SRPA):
    y = SRPA(x.length)
    for i in range(x.length):
        y[i] = sr_ift(x[i])
    return y
```

Multi-Residue Polynomial Gadgets

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".

A constructor for this type, generating an empty MRP with a given base can be of the form $M = MRP(\{q_1, q_2, \dots, q_k\})$, with q_i being moduli, or a constructor grouping several single-residue polynomials together with the form $M = MRP(\{srp_1, srp_2, \dots srp_k\}, \{q_1, q_2, \dots, q_k\})$, with srp_i being a single-residue polynomial, and q_i being its appropriate modulus.

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".

A constructor for this type, generating an empty MRS with a given base can be of the form $M = MRS(\{q_1, q_2, \dots, q_k\})$, with q_i being moduli, or a constructor grouping several single-residue scalars together with the form $M = MRS(\{srs_1, srs_2, \dots srs_k\}, \{q_1, q_2, \dots, q_k\})$, with srp_i being a single-residue scalar, and q_i being its appropriate modulus.

Multi-Residue Arithmetic Gadgets

Note that these gadgets assume that both operands have the same base. Supplied is a pseudocode describing an equivalent implementation in terms of baseline operations.

```
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
```

```
def mr_zeros(target_base : set, N : integer):
      z = MRP(target_base)
      for q in target_base:
            z[q] = sr_zeros(N)
      Return z
```

MRP Residue Manipulation Gadgets

Pseudocode:

```
def mr_append_srp (x: MRP, a: SRP, q_a):
      target_base = x.base + {q_a}     # set union
      z = MRP(target_base)
      z[q_a] = a
      for q in x.base:
            z[q] = x[q]
      return z
```

```
def mr_union (x: MRP, y: MRP):
      # assuming mutually exclusive bases for x and y
      target_base = x.base + y.base     # set union
      z = MRP(target_base)
      for q in x.base:
            z[q] = x[q]
      for q in y.base:
            z[q] = y[q]
      return z
```

```
def mr_subset (x: MRP, subbase : set):
      # assume subbase is a subset of x.base
      z = MRP(subbase)
      for q in subbase:
            z[q] = x[q]
      return z
```

Fast Base Conversion

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

- 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.

Pseudocode:

```
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
```

CKKS Rescale using Fast Base Conversion

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

- 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.

Pseudocode;

```
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
```

Multi-Residue Polynomial Array Gadgets

MRP Array (MRPA)

This is an array of MRPs. Each MRP may have a different number of residues, and a different moduli base. The array *length* can be accessed as a subfield.

A constructor for this type, generating an empty MRPA (length 0) can be of the form $MA = MRPA(\ )$ or a constructor grouping several MRPs together with the form $MA = MRPA(\{mrp_1, mrp_2, \dots, mrp_k\})$, with mrp_i being a multi-residue polynomial.

`x.append(y)` is pseudocode for in-place appending the MRP y in at the end of the MRPA x.

Multi-Residue Polynomial Array Gadgets

MRPA Dot-Product

Receives two MRP Arrays and returns single MRP, which is the dot-product operation between the two arrays.

Pseudocode:

```
def dotproduct (x: MRPA, y: MRPA):
     # assume x.length == y.length
```

```
      # assume that x[i].base = y[j].base for any i,j
      target_base = x[0].base
      N = x[0][0].ring_dimension
      z = MRP_zeros(target_base, N)
      for i in range(x.length):
            temp = mr_mulp(x[i],y[i])
            z = mr_addp(z, temp)
      return z
```

CKKS/BGV/BFV Digit Decomposition for Hybrid Key-Switching

Receives as input:

- A coefficient-representation MRP in RNS base representing modulus Q
- A list of *d* mutually exclusive moduli bases, each representing modulus $\tilde{Q}_i$ such that $Q = \prod_i \tilde{Q}_i$
- RNS base representing the temporary modulus P.

Returns as output:

- An MRPA, of length *d,* such elements *i* is an MRP that is a base extension to modulus QP from base $\tilde{Q}_i$

Pseudocode:

```
def dig_decomp (x: MRP, digit_bases : list of sets, Pbase :
set):
      target_base = x.base + Pbase    # set union
      d = len(digit_bases)
      z = MRPA(d)
      for i in range(d):
            temp = mr_subset (x, digit_bases[i])
            z[i] = FastBaseConvert (temp, target_base)
      return z
```

GSW/RLWE External Product

Note that the following gadget uses the optional instructions 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).

Pseudocode:

```
def GSWRLWEExtProd (gsw: SRPA, rlwe_in: SRPA):
    decomp_rlwe_0 = gadget_decomp(rlwe_in[0], B, l)
    decomp_rlwe_1 = gadget_decomp(rlwe_in[1], B, l)
    decomp_rlwe_0_ft = sr_batch_ft(decomp_rlwe_0, l)
    decomp_rlwe_1_ft = sr_batch_ft(decomp_rlwe_1, l)
    rlwe_tmp = SRPA(2)
    rlwe_tmp[0] = sr_zeros(N)
    rlwe_tmp[1] = sr_zeros(N)
    for level in range(l):
        rlwe_tmp[0] = sr_add(rlwe_tmp[0],
                             sr_mul(decomp_rlwe_0_ft[l],
                                    gsw[4*l]))
        rlwe_tmp[1] = sr_add(rlwe_tmp[1],
                             sr_mul(decomp_rlwe_0_ft[l],
                                    gsw[4*l+1]))
        rlwe_tmp[0] = sr_add(rlwe_tmp[0],
                             sr_mul(decomp_rlwe_1_ft[l],
                                    gsw[4*l+2]))
        rlwe_tmp[1] = sr_add(rlwe_tmp[1],
```

```
                                sr_mul(decomp_rlwe_1_ft[l],
                                       gsw[4*l+3]))
    return sr_batch_ift(rlwe_tmp)
```

Optional Operations

These are operations that are not implementable using the baseline operations, but that some hardware might choose to implement. Optional operations provide an avenue for defining direct, optimized hardware implementations of scheme-specific operations.

As mentioned in the section on basic data types, we will often refer to polynomials and scalars according to their first metadata field, namely with the descriptor “integer” or “non-integer”. For the purposes of distinguishing the optional operations defined here from the required baseline operations, we use the “non-integer” descriptor to indicate that all the inputs and outputs are non-integer polynomials and non-integer scalars. Please refer to the section on basic data types for the definitions of the terms “non-integer polynomial” and “non-integer scalar”.

Non-integer Polynomial Addition

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

f = sr_addp(a, b) $\rightarrow$ $f_i = a_i + b_i$

Non-integer Polynomial Scalar Addition

Takes as input a non-integer polynomial *a* and a scalar *s*, and returns a single non-integer polynomial. The operation adds *s* to every component of *a*. For *a* in the coefficient representation, the operation adds *s* to the first coefficient of *a* and returns the resulting polynomial.

f = sr_addps(a, s) $\rightarrow$ $f_i = a_i + s$
f = sr_addps_coeff(a, s) $\rightarrow$ $f_0 = a_0 + s$, $f_{i>0} = a_i$

Non-integer Polynomial Negation

Takes as input a single integer polynomial, *a*, and returns a single integer polynomial. The return value is the component-wise negation of each component of *a*.

f = sr_negp(a) $\rightarrow$ $f_i = -a_i$

Non-integer Polynomial Subtraction

Takes as input two non-integer polynomials, *a* and *b*, and returns a single non-integer polynomial. The return value is the component-wise difference of *a* and *b.*

f = sr_subp(a, b) → $f_i = a_i - b_i$

Non-integer Polynomial Scalar Subtraction

Takes as input a non-integer polynomial *a* and a scalar *s*, and returns a single non-integer polynomial. The operation subtracts *s* from every component of *a*. For *a* in the coefficient representation, the operation subtracts *s* from the first coefficient of *a* and returns the resulting polynomial.

f = sr_subps(a, s) → $f_i = a_i - s$
f = sr_subps_coeff(a, s) → $f_0 = a_0 - s$, $f_{i>0} = a_i$

Non-integer Polynomial Multiplication

Takes as input two non-integer polynomials, *a* and *b*, and returns a single non-integer polynomial. The return value is the component-wise product of *a* and *b.*

f = sr_mulp(a, b) → $f_i = a_i \times b_i$

Non-integer Polynomial Scalar Multiplication

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

f = sr_mulps(a, s) → $f_i = a_i \times s$

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 (denoted *sr_ifft*) 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.

Coefficient Extraction

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 *i*th component of *p*.
In pseudocode appears as `x = p[i]` or `x = CoeffExtract(p,i)`

Coefficient Assignment

Takes as input a polynomial *p*, an integer index *i*, and a value *val* and returns a polynomial with updated component *val* at position *i*.

In pseudocode, we write this `x=CoeffAssign(p,i,val)`, or `p[i]=val` in case in-place assignment is meant.

Torus Modular Reduction

Takes as input a non-integer polynomial *p* and a real number *c* and returns a polynomial *p'* with coefficients in the range *[c-0.5, c+0.5)* equal to those of *p* up to the addition of an integer polynomial.

Hardware may restrict the possible values of *c* or even support only one value (typical choices are 0 and 0.5).

Sample Extraction

This operation is only used by the TFHE and FHEW schemes. It takes as input an RLWE cipher *(a, b)* and returns an LWE cipher obtained by taking the first coefficient of *a*, followed by the n-2 opposites of the last coefficients of *a* in reverse order, where n is the LWE dimension, followed by the first coefficient of *b*.

Gadget Decomposition

This operation is only used by 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, …, 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 supporting the TFHE scheme should support Gadget Decomposition for at least one value of (l, b, K, E) compatible with the other supported parameters.

CKKS Bootstrapping

This tentative example optional operation promises to perform the bootstrapping for a given security level (e.g. 128bits classical), precision (e.g. 25bits) and probability of bootstrapping failure (e.g. 2^{-80}). All parameter values provided below are given as an example only. Much more specification would be needed to fully instantiate this idea as an Optional Operator. Also note that this operation takes as inputs and provides as outputs multi-residue polynomial arrays (MRPAs); for the definition of these objects, please see the section of the same name later in this document.

Inputs:

- ct_in - MRPA of length 2 - the input ciphertext to be bootstrapped, with each of the two contained MRPs being with a base representing a modulus Q_0 of 60bits.
- Aux_Data - MRPA of some length, including key switching keys and plaintext polynomials required for the operation.

Output:

- ct_out - MRPA of length 2 - the refreshed ciphertext, with each of the two contained MRPs being with a base representing a modulus Q_r of 420bits.

Information that should flow upward

A hardware solution should provide at least a minimum advertisement of its capabilities so that a compiler can make choices in code generation. At minimum, the following information must be provided in IR form for use by a relevant compiler:

- Ring dimension
- RNS field size
- Maximum length of modulus chain
- A list of available Gadgets (subset of the full list specified in this standard)
- Optional operations supported, if any (again - a subset of what the standard defines)

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 to be used in executing the code in an IR instance.
- List of primes for RNS representation to be used.
- Whether a polynomial is in coefficient form or evaluation form
- Precision of coefficients
- Prime congruence for the list of primes
- Instruction sequence

Future Enhancements

This section lays out some ideas for future additions to the specification, but makes no demands on the current specification. We welcome the input of all participants in developing these ideas more fully!

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.

Explicit Parallelism

It may be useful to allow downstream instruction sequences to be expressed with explicit parallelism opportunity, along the lines of Very Long Instruction Word (VLIW) architectures such as the original Intel® Itanium™ instruction bundling approach. The idea would be to allow a compiler to group or bundle instructions together, and allow hardware the option of following that

guidance in part or in full as a way of simplifying on-board scoreboarding and dependency analysis.

Memory Hierarchy Upward Advertisement

An advantage that FHE programs have over traditional programs that exhibit dynamic control flow is that all data access patterns are fully knowable at compile time. Thus it seems very likely that compiler action could mitigate the need for additional hardware such as cache tags and set associativity logic that are typically used to optimize memory placement of data. However, this puts the onus for such optimization squarely on the compiler. Example: the re-use of a polynomial in several instructions that appear nearby in execution order of a program presents an opportunity to keep that polynomial “handy” so as to not repeatedly move it within the memory hierarchy. Compilers are quite good at managing memory in this way, while hardware (in the absence of caches) is not. However, a compiler would need to understand the memory hierarchy provided by hardware in order to manage it effectively.

We may want to include upstream advertising of memory hierarchy structure to enable compilers to manage hierarchy use in optimizing for locality of reference. Such advertisement would likely include a description of the size and shape of each layer of accelerator memory structure, latencies and bandwidths between those layers, and the graph of connectivity among those layers. A syntax will be needed here that’s concise but detailed enough.

Additional gadget class

In addition to the simple and complex gadget classes already specified, we also acknowledge the potential for another class of gadgets we would tentatively describe as “approximate”. These would be gadgets for which the proposed implementation in terms of baseline/optional operations would only be required to be equivalent to the semantics of the gadget as described. However, in order to make this definition precise, we need to specify what we mean by “equivalent” in this context.

New gadgets

During the Hardware Breakout Session at the 8th HomomorphicEncryption.org Standards Meeting, CryptoLab expressed interest in contributing an example gadget for RNS GSW×RLWE multiplication. At the same meeting, the Belfort team also indicated their interest in adding a gadget for a batched key-switching plus programmable bootstrapping operation. We envision including such gadget specifications in future versions of the standard.

PRNG specification

There is general agreement on the utility of hardware PRNGs in this context. However, given the need to coordinate between hardware and software, it is unclear how best to specify these

and appropriately propagate that specification from hardware through to software. The group should explore any existing standards and decide on an approach.

Appendix: Syntax (Under Development)

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

Examples towards developing the syntax (Under Development)

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);
    }
}
```

Batched TFHE Key-Switching + Programmable Bootstrapping (KS-PBS)

Motivation: Existing FPGA implementations of TFHE benefit from working at a very high level and on batches of ciphers to optimize data pipelining. The TFHE scheme is particularly well-suited for that as most workflows can be formulated in a way where most of the runtime is due to a well-defined sequence of instructions, made of key-switching and PBS operations. This gadget would provide a single black-box ‘batched key-switching and PBS’ instruction with a well-defined interface.

Parameters (example value):

```
LWE_WORD_SIZE (12 bits)
LWE_DIMENSION (938)
RLWE_WORD_SIZE (32 bits)
RLWE_DIMENSION (2048)
PBS_DECOMPOSITION_LEVELS (1)
PBS_LOG_DECOMPOSITION_BASIS (23)
```

```
KS_DECOMPOSITION_LEVELS (2)
KS_LOG_DECOMPOSITION_BASIS (15)
MODULUS (1 << 64)
```

Description:

Inputs:

- batch_size (unsigned integer)
- Set of batch_size LWE ciphers representing the input data
- Set of batch_size RLWE ciphers representing (plaintext or encrypted) LUTs
- Key-switching key
- Bootstrapping key

Output: Set of batch_size LWE ciphers representing the output data

High-level description:

1. Run LWE key-switching on the input LWE ciphers.
    1. 2. Run the TFHE PBS on the output of key-switching.

High-level pseudocode:

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

void ks(const size_t batch_size,
        const lwe_coeff_t* const lwe_location,
        const rlwe_coeff_t* const ks_key_location,
        const lwe_coeff_t* lwe_output)
{
  for (unsigned int i = 0; i < batch_size; ++i) {
    const lwe_coeff_t
lwe_decomposed[KS_DECOMPOSITION_LEVELS][LWE_DIMENSION];
    gadget_decomposition(lwe_input + i*SIZE_LWE, lwe_decomposed,
                         KS_DECOMPOSITION_LEVELS,
                         KS_LOG_DECOMPOSITION_BASIS);
    lwe_coeff_t lwe_temp[LWE_DIMENSION];
    for (unsigned int j = 0; j < KS_DECOMPOSITION_LEVELS; ++j) {
      for (unsigned int k = 0; j < LWE_DIMENSION; ++k) {
        lwe_temp[k] = sr_addp(lwe_temp[k],
                              sr_mulp(lwe_decomposed[j][k],
ks_key_location + j*LWE_DIMENSION + k, 2*RLWE_DIMENSION),
                              2*RLWE_DIMENSION);
      }
    }
```

```
  }
}

void ks_pbs(const size_t batch_size,
            const lwe_coeff_t* const lwe_location,
            const poly_t& lut,
            const rlwe_coeff_t* const ks_key_location,
            const poly_t* const pbs_key,
            const poly_t* rlwe_temp,
            const poly_t* gsw_temp,
            const lwe_coeff_t* lwe_temp,
            const lwe_coeff_t* lwe_output)
{
     for (unsigned int i = 0; i < batch_size; ++i) {
          ks(lwe_location + i*LWE_DIMENSION, ks_key_location,
     lwe_temp);
          pbs(lwe_temp, lut_location, pbs_key, rlwe_temp,
              gsw_temp, lwe_output + i*SIZE_LWE);
     }
}
```