TFHE-rs: A (Practical) Handbook

First Edition

Mathieu Ballandras, Mayeul de Bellabre, Loris Bergerat, Charlotte Bonte, Carl Bootland, Benjamin R. Curtis, Jad Khatib, Jakub Klemsa, Arthur Meyre, Thomas Montaigu, Jean-Baptiste Orfila*, Nicolas Sarlin, Samuel Tap, David Testé

Zama
firstname.lastname@zama.ai,
*jb.orfila@zama.ai

February 27, 2025

Abstract

This document presents TFHE-rs, Zama's Rust library implementing an enhanced variant of TFHE. TFHE is a well-known fully homomorphic encryption (FHE) scheme that enables fast bootstrapping for small messages. TFHE-rs introduces advanced encoding techniques to support larger message sizes and employs ciphertext compression to mitigate expansion overhead. To provide a complete foundation, we first recall the necessary cryptographic concepts required to understand TFHE. We then present a comprehensive overview of TFHE-rs, detailing part of its core algorithms and the methods used to construct homomorphic integer ciphertexts—capable of encrypting, for example, a u64—thus significantly expanding the range of possible message spaces. The document systematically presents the operations available on these homomorphic integer types, integrating references to the TFHE-rs API. Additionally, we offer extensive benchmarking results that justify the design choices made in the library, evaluating the latency of all integer operations introduced in this work. Finally, we present recommended parameter sets that ensure secure and efficient execution of these algorithms across a range of failure probabilities.

Contents

1 Introduction

Fully homomorphic encryption (FHE) enables arbitrary computations on encrypted data without requiring decryption. Because the data remains encrypted during computation, a data owner can securely outsource processing to a third party without concern that it will be misused, shared with unauthorized parties, or exposed in a security breach. The first encryption scheme that could be considered *fully homomorphic* was introduced by Gentry [Gen09]. To achieve this, Gentry introduced the notion of *bootstrapping*. Bootstrapping is an operation that, informally, refers to homomorphically evaluating the decryption circuit to remove one encryption layer while simultaneously adding another.

This process is necessary because each ciphertext contains some noise, which increases with each computation. If left unchecked, the noise eventually grows too large, resulting in incorrect decryption. Bootstrapping replaces a highly noisy encryption layer with a less noisy one, allowing further computations while preventing excessive noise accumulation. All practical FHE schemes follow this same blueprint: ciphertexts inherently contain noise, and bootstrapping is used during computation to allow more operations to be realized. This applies to the TFHE scheme as well [CGGI16a].

TFHE is a fully homomorphic encryption (FHE) scheme based on variants of the Learning with Errors (LWE) problem. Without going into details, an LWE-based ciphertext consists of a vector of uniformly chosen modular integers (the mask) and an additional integer (the body), which combines the secret key, the message, the mask, and some noise. Using these ciphertexts, TFHE supports the evaluation of small lookup tables via the so-called programmable bootstrapping (PBS) operation. The efficiency of PBS is one of the defining features of TFHE, enabling efficient computation of functions over ciphertexts. However, TFHE has notable drawbacks, chief among them being its limited support for small message sizes (typically 1 to 8 bits) and its restriction to operating on a single message at a time. As a result, a single PBS operation alone offers limited utility. Another drawback is that TFHE ciphertexts have a significantly larger expansion factor—the ratio between ciphertext size and plaintext size—compared to other FHE schemes that can pack and process multiple data values simultaneously.

TFHE-rs is a Rust-based library developed by Zama that implements its variant of the TFHE scheme. Zama's TFHE variant is designed to address these drawbacks. Specifically, it supports encrypted integer arithmetic for *large integers*, such as `u16, u32, u64`. At a high level, this is achieved by grouping individual LWE ciphertexts, each encrypting a small number of bits, and using an encoding that preserves the structure of the large integer. Using this approach as a building block, the library provides various algorithms for performing full integer arithmetic (and more) on encrypted data. A graphical representation of this concept is shown in Figs. 1 and 2.

Figure 1: Representation of an LWE ciphertext used in the TFHE scheme. Here, the yellow blocks represent the random mask terms modulo q used in encryption (denoted a_i in future sections), and the grey block represents the body term which contains the encrypted message bits (denoted b in future sections). The number of yellow blocks (denoted n in future sections), and the chosen modulus q are directly related to performance and security. Typically, these ciphertexts encrypt a small integer, say 1-8 bits, supporting up to `u8` and enable encrypted arithmetic over small values.

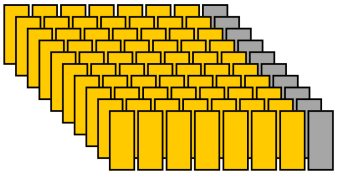

Figure 2: Representation of an integer ciphertext as constructed in the TFHE-rs library from a number of LWE ciphertexts. In order to construct these ciphertexts, we need several LWE ciphertexts as presented in Fig. 1. The number of LWE ciphertexts required to build an integer is directly related to performance. These integer ciphertexts can encrypt integers of size up to `u256` on which we can perform full integer arithmetic.

One of the main goals of TFHE-rs is to abstract the complexity of cryptography, ensuring that users do not need to worry about parameters and thus the security, correctness, or robustness, as these aspects are handled internally by the library. As a result, TFHE-rs abstracts the complexity of interacting with ciphertext types, providing a seamless user experience. Specifically, users interact directly with encrypted values, such as `u64`, and can perform various encrypted operations (e.g., addition or comparison) without handling the individual 'chunks' composing the integer ciphertext.

In Listing 1, the two `u8` values, `clear_1` and `clear_2`, are generated, encrypted, and then subjected to a variety of operations, all of which are described in later sections of this document. Each ciphertext consists of four LWE ciphertexts (blocks), but this complexity is entirely hidden from the user. The interaction between the four LWE ciphertexts forming `ct_1` and the four LWE ciphertexts forming `ct_2` is handled entirely by the TFHE-rs API.

```
use tfhe::prelude::*;
use tfhe::{generate_keys, set_server_key, ConfigBuilder, FheUint64};

fn main() {
  let config = ConfigBuilder::default().build();

  // Evaluation keys generated on the client-side
  let (client_key, server_key) = generate_keys(config);
  set_server_key(server_key);

  // Plaintext values generated on the client-side
  let clear_1 = 35u64;
  let clear_2 = 7u64;

  // Client-side Encryption
  let ct_1 = FheUint64::encrypt(clear_1, &client_key);
  let ct_2 = FheUint64::encrypt(clear_2, &client_key);

  // Take a reference to avoid moving data when doing the computation
  let ct_1 = &ct_1;
  let ct_2 = &ct_2;

  // Server-side computation using Rust's built-in operators
  let add = ct_1 + ct_2;
  let mul = ct_1 * ct_2;
  let shl = ct_1 << ct_2;

  // Client-side decryption and verification of proper execution
     ↪          (identical for other operations)
  let decrypted_add: u64 = add.decrypt(&client_key);
}
```

Listing 1: Example of the TFHE-rs high-level API

The individual LWE ciphertexts in the above example have a large expansion factor (in the tens of thousands), meaning that homomorphic integer types are significantly larger than their plaintext equivalents. Therefore, in applications requiring the storage of many such integers, a compression algorithm for these large LWE ciphertexts is beneficial. The TFHE-rs library includes a compression algorithm that can pack multiple homomorphic integers (each consisting of several LWE ciphertexts) into a *single* GLWE ciphertext, significantly reducing the ciphertext expansion factor.

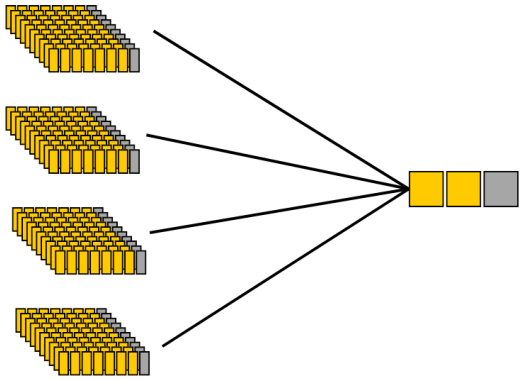

Figure 3: Representation of several integer ciphertexts being compressed into a single GLWE ciphertext

1.1 Document Goals

The goal of this document is to give a thorough overview of the TFHE-rs library and the algorithms contained within it. In particular, we outline the suite of algorithms available in the TFHE-rs library for both small message sizes (as represented in Fig. 1) and large message sizes (as represented in Fig. 2). Furthermore, this document is meant for developers and researchers using the TFHE-rs library to:

1. Understand the algorithms used in the TFHE scheme.
2. Introduce the integer arithmetic implemented in the TFHE-rs library.
3. Provide all the necessary details to build a new backend for the TFHE-rs library (for example, for specific hardware) and then benchmark it against other backends.

We focus in particular on describing operations, and the resulting algorithms that are implemented. The document does not cover correctness or noise arguments but instead points to existing arguments in the literature where relevant.

1.2 Document Structure

In Section 2, we outline notation, the underlying hard problems which guarantee security, and the low-level operations which make up the core of the TFHE cryptosystem. Namely, we cover basic ciphertext structures, encoding, encryption, decryption, and decoding algorithms, and the various operations supported in the core TFHE scheme.

In Section 3, we first introduce an encoding for large integers that precedes the TFHE encoding and encryption steps, resulting in something referred to as the *homomorphic integer*. Next, we outline all of

the supported algorithms over these homomorphic integer types, including basic arithmetic operations, bit-wise (logical) operations, comparisons, etc.

In Section 4, we look at a notion of ciphertext compression for the TFHE scheme, outlining the compression and decompression algorithms.

In Section 5, we cover a variety of benchmarks for the operations supported in TFHE-rs. In particular, we consider the latency for a variety of homomorphic integer sizes.

2 The TFHE scheme

Overview of TFHE. The TFHE fully homomorphic encryption scheme as implemented in the TFHE-rs library is a variant of the one introduced in [CGGI16b, CGGI17, CGGI20]. As previously mentioned, TFHE differentiates itself from the other (G)LWE-based FHE schemes because it supports a *very efficient bootstrapping* technique which we call the *programmable bootstrap* (PBS) operation. This operation is a bootstrap in that it resets the amount of noise in a ciphertext to a fixed level but it is also *programmable*, which means that at the same time, it can apply a function (via a lookup table evaluation) to the underlying data. In this sense, it is also sometimes called a *functional* bootstrapping. Depending on how the data is encoded in the ciphertext, one may either evaluate any function (which limits the usable message space to half) or be restricted to a so-called negacyclic function.

Unlike many other FHE schemes that use addition and multiplication as the fundamental operations from which to build more complicated functionality, there is no native multiplication in TFHE. However, several methods using the PBS allow the multiplication of two TFHE ciphertexts. It is the PBS operation that enables TFHE to go from being an additively homomorphic scheme to a fully homomorphic one.

TFHE was not the first FHE scheme to operate this way; it was originally proposed as an improvement of *FHEW* [DM15], a *GSW* [GSW13]-based scheme featuring fast bootstrapping for homomorphic boolean gate evaluation. Apart from improving the bootstrapping of FHEW, TFHE also introduced new techniques in order to support more functionalities and to improve the homomorphic evaluation of complex circuits. The efficiency of TFHE comes in part from the choice of a small ciphertext modulus which allows to use CPU native types to represent a ciphertext both in the standard domain and in the Fourier domain.

Notation. Let q be a positive integer, which will denote the ciphertext modulus. The ring of integers modulo q, $\mathbb{Z}/q\mathbb{Z}$, is denoted by $\mathbb{Z}_q$, and $\mathfrak{R}_{q,N}$ represents the polynomial quotient ring $\mathbb{Z}_q[X]/(X^N+1)$, where N is a power of two. Elements of $\mathbb{Z}_q$ are given by lowercase letters, such as a, and elements of the ring $\mathfrak{R}_{q,N}$ are given by uppercase letters, such as A. The coefficients of a polynomial A are given by $(a_0, a_1, \cdots, a_{N-1})$. Vectors are represented by bold letters: a vector of elements in $\mathbb{Z}_q$ is given by, e.g., $\mathbf{a}$ and a vector of elements in $\mathfrak{R}_{q,N}$ is given by, e.g., $\mathbf{A}$. Rounding x up, down, and to the closest integer is given by $\lceil x \rceil$, $\lfloor x \rfloor$, and $\lfloor x \rceil$. Reducing x modulo q is given by $x \bmod q$, and we extend this operation element-wise for vectors and coefficient-wise for polynomials. The dot product of two vectors is given by $\langle \mathbf{a}, \mathbf{b} \rangle = \sum_i a_i \cdot b_i$ and, similarly to reduction modulo q, this notation extends to vectors of polynomials. We denote the uniform distribution over a set S by $\mathcal{U}(S)$, and a Gaussian distribution with mean μ and standard deviation σ as $\mathcal{D}_{\mu,\sigma}$, in the case that $\mu = 0$ we simply write $\mathcal{D}_\sigma$.

We use the notation $a \leftarrow \mathcal{A}(\,\cdot\,)$ for a being the result of some deterministic algorithm $\mathcal{A}$ which can take any number of arguments. Similarly, we use the notation $a \hookleftarrow \mathcal{A}(\,\cdot\,)$ to denote a as the output of a probabilistic algorithm $\mathcal{A}$, which again can take a number of arguments.

In this document, we use different notation than the original TFHE papers [CGGI16b, CGGI17, CGGI20]. In those papers, the message and ciphertext spaces are expressed by using the real torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$. In the TFHE library [CGGI16c], $\mathbb{T}$ is typically implemented by using native arithmetic modulo 2^{32} or 2^{64}, which means that they work on $\mathbb{Z}_q$ (with $q = 2^{32}$ or $q = 2^{64}$). This is why we prefer to use $\mathbb{Z}_q$ instead of $\mathbb{T}$, as already adopted in [CJL+20]. It is made possible because there is an isomorphism between $\mathbb{Z}_q$ and $\frac{1}{q}\mathbb{Z}/\mathbb{Z}$ as explained in [BGGJ20, Section 1].

Outline. We begin by introducing the hard lattice problems on which TFHE is based, explaining why low-level ciphertexts take their specific form. Next, we describe these low-level ciphertexts and how they can be grouped to construct other basic ciphertext types used in the TFHE scheme, along with a simple method for encoding small integers. We then present basic operations that can be performed

on these ciphertexts. Following this, we introduce decomposition and explore several variants of the key-switching operation. Finally, we cover all the building blocks of the original Programmable Bootstrapping (PBS) and discuss some of its variants.

2.1 Hard Lattice Problems: the LWE and GLWE Assumptions

The security of the TFHE scheme is based on a family of hard problems, namely the Learning With Errors (LWE) problem of Regev [Reg05, Reg09] and its generalized variants [SSTX09, LPR10, BGV12, LS15]. These problems are the most commonly used foundations for building FHE schemes. However, other problems have also been explored [vDGHV10, LATV12, BIP+22a], such as the NTRU problem.

Informally, the LWE problem can be thought of as solving a system of noisy linear equations. The problem comes in two flavours: a search problem, and a decision problem. The two flavours have been shown to be equivalent in terms of security [ACPS09, MM11, MP12].

Definition 1 (Learning With Errors (LWE)) *Let $n \in \mathbb{N}$ be a positive integer, which we call the LWE dimension. Let $q \in \mathbb{N}$ be a positive integer, which we call the ciphertext modulus. Let $\mathbf{s} = (s_0, \cdots, s_{n-1}) \in \mathbb{Z}_q^n$ be a vector, which we call the secret, where for each $0 \leq i < n$, s_i is sampled from some distribution $\mathcal{D}$. We call $\mathcal{D}$ the secret distribution. Further, let χ be another distribution which we call the error distribution. We define a sample from the learning with errors distribution $\mathsf{LWE}_{q,n,\chi,\mathcal{D}}$ to be of the form $(\mathbf{a}, b = \sum_{i=0}^{n-1} a_i \cdot s_i + e) \in \mathbb{Z}_q^{n+1}$, where $\mathbf{a} = (a_0, \ldots, a_{n-1}) \hookleftarrow \mathcal{U}(\mathbb{Z}_q)^n$, meaning that each a_i is sampled uniformly and independently from $\mathbb{Z}_q$, and the error or noise $e \in \mathbb{Z}_q$ is sampled from χ.*

The decisional $\mathsf{LWE}_{q,n,\chi,\mathcal{D}}$ *problem [Reg05] consists in distinguishing independent samples from $\mathsf{LWE}_{q,n,\chi,\mathcal{D}}$ from the same amount of samples from the uniform distribution $\mathcal{U}(\mathbb{Z}_q)^{n+1}$.*

The search problem *consists of finding the secret $\mathbf{s}$ given an arbitrary polynomial number of samples from the learning with error distribution $\mathsf{LWE}_{q,n,\chi,\mathcal{D}}$.*

The hardness of both of these problems depends on the parameters (q, n) and on the noise distribution χ and the secret key distribution $\mathcal{D}$. The hardness of an LWE instance is described by a value λ which is called the *security level*. For an attacker to break an LWE instance with a security level λ, they should perform at least 2^λ operations. More precisely, breaking an LWE instance with a security level λ should take as many (or more) computational resources than those required to break a block cipher with a λ-bit key[1]. In practice, $\lambda = 128$ is the default value to guarantee long-term security of a particular instance.

While the LWE problem is a strong candidate for a hard problem, it results in a significant expansion factor between the unencrypted and encrypted data. When multiple values are to be encrypted, one can decrease this expansion factor by using a generalized version of the LWE problem which uses more complex mathematical structures than just integers. Below, we define the General Learning With Errors (GLWE) problem from [BGV12, LS15] (also known as the Module Learning With Errors problem) on the ring $\mathfrak{R}_{q,N}$ defined as $\mathfrak{R}_{q,N} = \mathbb{Z}_q[X]/\langle X^N + 1 \rangle$ with N a power of two. Elements of this ring can be written as polynomials of at most degree $N-1$ having integer coefficients. Addition and multiplication follow standard polynomial operations modulo q. However, after multiplication, the result is reduced to a degree at most $N-1$ by taking the remainder modulo the polynomial $X^N + 1$. Using the relation $X^N = -1$, any term with an exponent greater than N can be reduced accordingly. See also Remark 14.

Definition 2 (General Learning With Errors (GLWE)) *Let $N \in \mathbb{N}$ be a power of two, which we call the polynomial size. Let $k \in \mathbb{N}$ be a positive integer, which we call the GLWE dimension. Let*

[1] https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-criteria/security-(evaluation-criteria)

$q \in \mathbb{N}$ be a positive integer, which we call the ciphertext modulus. Let $\mathbf{S} = (S_0, \cdots, S_{k-1}) \in \mathfrak{R}_{q,N}^k$ be a vector, which we call the secret, where for each $0 \leq i < k$, $S_i = \sum_{j=0}^{N-1} s_{i,j} X^j$ is sampled from some distribution $\mathcal{D}$ which we call the secret distribution, and let χ be an error (noise) distribution. We define a sample from the general learning with errors distribution *$\mathsf{GLWE}_{q,N,k,\chi,\mathcal{D}}$ to be of the form $(\mathbf{A}, B = \sum_{i=0}^{k-1} A_i \cdot S_i + E) \in \mathfrak{R}_{q,N}^{k+1}$ where $\mathbf{A} = (A_0, \ldots, A_{k-1}) \hookleftarrow \mathcal{U}(\mathfrak{R}_{q,N})^k$, meaning that all the coefficients of A_i are sampled uniformly and independently from $\mathbb{Z}_q$, and the error (noise) polynomial $E \in \mathfrak{R}_{q,N}$ is such that all the coefficients are sampled from χ.*

The decisional $\mathsf{GLWE}_{q,N,k,\chi,\mathcal{D}}$ problem [LS15, BGV12] consists in distinguishing m independent samples from $\mathcal{U}(\mathfrak{R}_{q,N})^{k+1}$ from the same amount of samples from $\mathsf{GLWE}_{q,N,k,\chi,\mathcal{D}}$, where $\mathbf{S} \in \mathfrak{R}_{q,N}^k$ follows the distribution $\mathcal{D}$.

The search problem consists in finding $\mathbf{S}$ given an arbitrary polynomial number of samples from the learning with error distribution $\mathsf{GLWE}_{q,N,k,\chi,\mathcal{D}}$.

If we choose $N = 1$ and $k = n$, the GLWE distribution $\mathsf{GLWE}_{q,N,k,\chi,\mathcal{D}}$ is the same as the LWE distribution $\mathsf{LWE}_{q,n,\chi,\mathcal{D}}$. If we choose $N > 1$ and $k = 1$, the GLWE distribution is called the RLWE distribution [SSTX09, LPR10]. The GLWE problem can be seen as a generalization of both the LWE and the RLWE problems.

In general, the secret key distribution $\mathcal{D}$ is such that the polynomial coefficients are usually either sampled from a uniform binary distribution, a uniform ternary distribution, or a Gaussian distribution ([BJRW23, ACPS09]).

Remark 1 (Alternative hardness assumptions) *Some recent works adapt TFHE algorithms to operate on ciphertexts whose security is based on different hard problems. For example, the authors of [BIP+22b, XZD+23] adapt TFHE's PBS algorithm to work on NTRU ciphertexts rather than GLWE ciphertexts*

2.2 The Morphology of FHE Ciphertexts

In this section, we describe several kinds of low-level ciphertexts. The security of these ciphertexts relies on the hardness of the problems introduced in Definitions 1 and 2.

First, we recall the LWE and the GLWE ciphertexts which are the most common low-level type of ciphertexts. GLev and GGSW ciphertexts are then defined as special collections of GLWE ciphertexts.

2.2.1 LWE, RLWE & GLWE Ciphertexts

The most common type of ciphertexts in TFHE as described in [CGGI20] is the LWE ciphertext. The security of an LWE ciphertext relies on the LWE problem (Definition 1). Intuitively, an LWE ciphertext can be viewed as a sample from an LWE distribution with an encoded message added to its last component. We will come back to what is meant by an encoded message in Section 2.3, for now, what is important is that it is just some element of $\mathbb{Z}_q$.

Definition 3 (LWE Ciphertext) *Let $q \in \mathbb{N}$ be a positive integer, which we call the ciphertext modulus. Let $n \in \mathbb{N}$ be a positive integer, which we call the LWE dimension. Given an encoded message $\widetilde{m} \in \mathbb{Z}_q$ and a secret key $\mathbf{s} = (s_0, \cdots, s_{n-1}) \in \mathbb{Z}_q^n$, an* LWE *ciphertext of $\widetilde{m}$ under the secret key $\mathbf{s}$ is defined as the tuple:*

$$\mathsf{ct} = \left(a_0, \cdots, a_{n-1}, b = \sum_{i=0}^{n-1} a_i \cdot s_i + \widetilde{m} + e \right) \in \mathsf{LWE}_{\mathbf{s}}(\widetilde{m}) \subseteq \mathbb{Z}_q^{n+1}, \tag{1}$$

where each a_i can be viewed as being a sample from the uniform distribution over $\mathbb{Z}_q$ and e is a noise (error) in $\mathbb{Z}_q$ which can be viewed as having been sampled from an error distribution χ. The parameter $n \in \mathbb{N}$ represents the number of elements in the LWE *secret key.*

Where the usage of "can be viewed as being a sample from" is to capture not just freshly encrypted ciphertexts (for which the a_i and e are actually sampled from the respective distributions), but also ciphertexts that have been processed in some way. For the elements a_i it is important for security that they look uniformly random. For the error term e, the distribution χ will change depending on where the ciphertext has come from and what operations it has performed on it. It is important to track theoretically this distribution while doing computations for correctness. In TFHE, low-level ciphertexts do not need to be explicitly tagged with this information, provided computations follow predefined patterns, known as atomic patterns, which are outlined in Section 2.7.5. These patterns ensure correct computation up to a given failure probability associated with the parameter set.

As mentioned earlier, inside many TFHE algorithms, we use another type of ciphertext, this time using a sample from a GLWE distribution to blind an encoded message. We naturally call a ciphertext of this form a GLWE ciphertext, and it can be seen as a generalization of an LWE ciphertext. The security of a GLWE ciphertext relies on the hardness of the GLWE problem introduced in Definition 2. In Definition 4, we again use $\mathfrak{R}_{q,N} = \mathbb{Z}_q[X]/\langle X^N + 1\rangle$ with N a power of two.

Definition 4 (GLWE Ciphertext) *Given an encoded message $\widetilde{M} \in \mathfrak{R}_{q,N}$ and a secret key $\mathbf{S} = (S_0, \cdots, S_{k-1}) \in \mathfrak{R}_{q,N}^k$, a* GLWE *ciphertext of $\widetilde{M}$ under the secret key $\mathbf{S}$ is a tuple:*

$$\mathsf{CT} = \left(A_0, \cdots, A_{k-1}, B = \sum_{i=0}^{k-1} A_i \cdot S_i + \widetilde{M} + E \right) \in \mathsf{GLWE}_{\mathbf{S}}(\widetilde{M}) \subseteq \mathfrak{R}_{q,N}^{k+1}$$

where each A_i is a polynomial in $\mathfrak{R}_{q,N}$ with coefficients which are sampled from the uniform distribution over $\mathbb{Z}_q$, and E is a noise (error) polynomial in $\mathfrak{R}_{q,N}$, with coefficients which are sampled from an error distribution χ (see also Remark 2). The parameter $k \in \mathbb{N}$ represents the number of polynomials in the GLWE *secret key and is called the GLWE dimension. We also call N the polynomial size of* CT *and q its ciphertext modulus.*

Remark 2 *In the above definition, we assumed that the noise polynomial has coefficients which all follow the same distribution. Occasionally, we will have ciphertexts for which this is not the case, then we would need to specify different distributions χ_i for each coefficient of E.*

Remark 3 (Independence of Noise) *Note that in the definition of a ciphertext, we do not require the coefficients of the noise polynomial to be independent. Ideally, they would have this property, but this is not always the case.*

If we set $N = 1$ and $k = n$ in Definition 4, we obtain the same type of ciphertext as in Definition 3. In the special case where $N = 1$, the security of the encryption relies on the hardness of the LWE problem (Definition 1) and not on the GLWE problem (Definition 2). By convention, throughout this paper, we will write an LWE ciphertext, an LWE secret key, and an integer message in lowercase, e.g. ct, $\mathbf{s}$, and m. On the contrary, we use upper case for a GLWE ciphertext, a GLWE secret key, and a polynomial message when $N > 1$, e.g. CT, $\mathbf{S}$, and M.

Another special case of Definition 4 is when we set $k = 1$ and $N > 1$. This ciphertext is called an *RLWE ciphertext.*

Remark 4 (Mask and Body of a Ciphertext) *In Definition 3, respectively Definition 4, we call the elements $(a_0, \cdots, a_{n-1})$, respectively $(A_0, \cdots, A_{k-1})$, the ciphertext mask while we call the remaining element, b, respectively B, the ciphertext body.*

Remark 5 (Fresh Ciphertext Compression) *Freshly encrypted (G)LWE ciphertexts can be compressed via the usage of a cryptographically secure pseudo-random number generator (CSPRNG), denoted by ψ, to derive the random mask terms $a_0, ..., a_{n-1}$ via evaluation of ψ on a seed $\rho \in \{0,1\}^\lambda$, e.g.*

$$(a_1, \cdots, a_{n-1}) \leftarrow \psi(\rho).$$

This means that a fresh LWE ciphertext $(a_1, \cdots, a_{n-1}, b)$ can be reduced to (ρ, b). The same technique can be applied to GLWE ciphertexts to generate the random mask terms $(A_1, \cdots, A_{k-1})$.

Remark 6 (Secret Key and Error Distributions) *In TFHE, we typically use a secret key distribution that samples coordinates/coefficients independently at random from a binary distribution. Other distributions such as ternary and even discrete Gaussian distributions can also be considered though typically only in the GLWE setting due to how bootstrapping is done in TFHE.*

For the error (or noise) distribution, the classical example is to use a discrete Gaussian distribution (coefficient-wise if GLWE) which has some relatively small variance. Furthermore, due to the central limit theorem, after computation, it is often the case that the noise inside a ciphertext has a distribution that is well modeled by a (discrete) Gaussian distribution, regardless of what the initial noise distribution was. For simplicity, the discreteness of the Gaussian is usually ignored and we treat them as continuous. This holds as long as the standard deviation of the discrete Gaussian distribution is large enough.

More recently, a second type of error distribution, which we call a TUniform distribution has been considered for encrypting fresh ciphertexts (i.e., ciphertexts which are the output of the encryption algorithm). This distribution is close to the uniform distribution on some interval $[-B+1, B]$ of length $2B$, except that we add $-B$ to its support and set the probability of sampling both $-B$ and B to be $1/4B$, ensuring the mean of this distribution is zero. This distribution has the advantage of being much easier to sample from which is crucial in the threshold encryption setting where the noise has to be generated in a secret-shared manner.

Remark 7 (Private and Public Key Encryption) *There are two possible ways to encrypt a ciphertext, depending on whether the system has been set up to be a private or public key scheme. In a private key scheme, only the holder of the secret key can encrypt an encoded message while in a public key scheme, anyone who knows the public key can encrypt an encoded message but only someone knowing the private key can decrypt. The typical way of going from a private key FHE scheme to a public one is for a number of encryptions of zero to be published as the public key. Then, to encrypt an encoded message using the public key one can take the sum of a random subset of the encryptions of zero and then add the encoded message to the body of the resulting ciphertext (plus some extra noise). For more details see Section 2.4.3.*

We present the (private key) encryption algorithm in Algorithm 1 for encrypting GLWE ciphertexts, the case of encrypting LWE ciphertexts can be seen as having $N = 1$ as $\mathfrak{R}_{q,1} = \mathbb{Z}_q$.

In the next definition, we show how to decrypt a GLWE ciphertext (and therefore also as a special case, an LWE ciphertext). We point out that after decryption we get an encoded message, thus there is typically a decoding step required to remove the noise/error before obtaining the data that was encrypted in the ciphertext.

Definition 5 (GLWE Decryption) *Given a GLWE secret key $\mathbf{S} = (S_0, \cdots, S_{k-1}) \in \mathfrak{R}_{q,N}^k$ and a GLWE ciphertext CT of $\widetilde{M}$ under the secret key $\mathbf{S}$ as defined in Definition 4, the decryption of CT is defined as:*

$$\begin{aligned} \overline{M} &= B - \sum_{i=0}^{k-1} A_i \cdot S_i \\ &= \widetilde{M} + E \in \mathfrak{R}_{q,N} \end{aligned} \tag{2}$$

Algorithm 1: $\mathsf{CT} \leftarrow \mathsf{Encrypt}\left(\widetilde{M}, \mathbf{S}; \chi_\sigma\right)$

Context: $\begin{cases} \chi_\sigma : \text{ a noise distribution with standard deviation } \sigma \text{ which depends on } k \cdot N \\ \mathcal{U}(\mathfrak{R}_{q,N}) : \text{ the uniform distribution over } \mathfrak{R}_{q,N} \end{cases}$

Input: $\begin{cases} \mathbf{S} = (S_0, \ldots, S_{k-1}) : \text{ the GLWE secret key} \\ \widetilde{M} \in \mathfrak{R}_{q,N} : \text{ an encoded message} \end{cases}$

Output: $\mathsf{CT} \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M}\right)$

for $0 \leq j \leq N-1$ **do**
  $e_j \hookleftarrow \chi_\sigma$
$E \leftarrow \sum_{j=0}^{N-1} e_j \cdot X^j$
for $0 \leq i \leq k-1$ **do**
  $A_i \hookleftarrow \mathcal{U}(\mathfrak{R}_{q,N})$
$B \leftarrow \sum_{i=0}^{k-1} A_i \cdot S_i + E + \widetilde{M}$
$\mathsf{CT} \leftarrow (A_0, \cdots, A_{k-1}, B)$
return CT

The decryption of a GLWE ciphertext is the modular addition between the encoded message $\widetilde{M}$ and the noise polynomial E. In [CGGI20], $\overline{M}$ is called the phase *of* CT *and is denoted $\phi(\mathsf{CT}, \mathbf{S})$.*

An algorithmic description of decryption is given in Algorithm 2.

Algorithm 2: $\overline{M} \leftarrow \mathsf{Decrypt}(\mathsf{CT}, \mathbf{S})$

Input: $\begin{cases} \mathbf{S} = (S_0, \ldots, S_{k-1}) : \text{ the GLWE secret key} \\ \mathsf{CT} = (A_0, \ldots, A_{k-1}, B) \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M}\right) : \text{ the GLWE ciphertext} \end{cases}$

Output: $\overline{M} \in \mathfrak{R}_{q,N}$: a noisy version of $\widetilde{M}$, called the phase of CT

$\overline{M} \leftarrow B - \sum_{i=0}^{k-1} A_i \cdot S_i$
return $\overline{M}$

One can view the phase as simply another encoding of the underlying data that is being encrypted. It may or may not be possible to extract this data from the new encoding because of the noise term E. This noise is in the lower bits of the encoded message so intuitively any encoding will want to place the actual message in the higher order bits. Choosing suitable parameters allows to be able to decode this noisy encoding with very high probability. More details on the encodings used in TFHE-rs are given in Sections 2.3 and 3.1.

We will sometimes use ciphertexts that are trivially encrypted. The next definition explains what a trivial encryption is but intuitively it is where all randomness is set to zero.

Definition 6 (Trivial Encryption) *Given an encoded message $\widetilde{M} \in \mathfrak{R}_{q,N}$ and a* GLWE *dimension k, a* trivial GLWE *encryption of $\widetilde{M}$ is defined as the tuple:*

$$\mathsf{CT}^{(\mathsf{Triv})} = \left(0, \cdots, 0, B = \widetilde{M}\right) \in \mathsf{GLWE}_{k,N}^{(\mathsf{Triv})}(\widetilde{M}) \subseteq \mathfrak{R}_{q,N}^{k+1},$$

where $\mathsf{GLWE}_{k,N}^{(\mathsf{Triv})}(\widetilde{M}) = \left\{\left(0, \cdots, 0, \widetilde{M}\right)\right\} \subseteq \mathsf{GLWE}_{\mathbf{S}}(\widetilde{M})$ for any $\mathbf{S} \in \mathfrak{R}_{q,N}^k$. Informally, $\mathsf{CT}^{(\mathsf{Triv})}$ can be seen as a GLWE encryption of $\widetilde{M}$ under any secret key $\mathbf{S} \in \mathfrak{R}_{q,N}^k$.

Of course, as the mask polynomials and the noise polynomial are set to zero, these 'ciphertexts' are not secure as there is no encryption. They are mainly used to simplify the notation in some algorithms referring to an input that can be either a ciphertext or a plaintext.

2.2.2 Lev, RLev & GLev Ciphertexts

Using Definition 4, we can build more complex types of ciphertexts that are required to define the public material used in some FHE algorithms. For instance, the keyswitching key in Algorithm 10 can be defined with the help of GLev ciphertexts. Informally, a GLev ciphertext is just a collection of related GLWE ciphertexts that come with two additional parameters: a decomposition base and a decomposition level. Note that we do not encrypt an encoded message here but a plaintext directly, as such these will only be of practical use if this plaintext is small compared to the ciphertext modulus q.

Definition 7 (GLev Ciphertext [CLOT21]) *Given a ciphertext modulus q, a decomposition base $\mathfrak{B} \in \mathbb{N}$ and a decomposition level $\ell \in \mathbb{N}$ such that $\mathfrak{B}^\ell$ divides q, a* GLev ciphertext *of a plaintext $M \in \mathfrak{R}_{q,N}$ under a GLWE secret key $\mathbf{S} \in \mathfrak{R}_{q,N}^k$ is defined as the following tuple of GLWE ciphertexts:*

$$\overline{\mathsf{CT}} = (\mathsf{CT}_0, \cdots, \mathsf{CT}_{\ell-1}) \in \mathsf{GLev}_{\mathbf{S}}^{\mathfrak{B},\ell}(M) \subseteq \mathfrak{R}_{q,N}^{\ell \times (k+1)} \tag{3}$$

such that for each $0 \leq j < \ell$,

$$\mathsf{CT}_j \in \mathsf{GLWE}_{\mathbf{S}}\Big(\frac{q}{\mathfrak{B}^{j+1}} \cdot M\Big) \subseteq \mathfrak{R}_{q,N}^{k+1}.$$

A GLev ciphertext with $N = 1$ is called a Lev *ciphertext* and in this case we typically write n in place of k for the size of the LWE secret key. A GLev ciphertext with $k = 1$ and $N > 1$ is called a RLev *ciphertext*.

Remark 8 *One can define a* GLev *ciphertext for a ciphertext modulus q for which $\mathfrak{B}^\ell$ does not divide q in a number of ways, for example by rounding each $\frac{q}{\mathfrak{B}^{j+1}} \cdot M$ before encrypting it. Such GLev ciphertexts behave in much the same way but with some extra noise terms appearing in the analysis. To keep things simple we will stick to the case for which $\mathfrak{B}^\ell$ divides q in this document.*

2.2.3 GSW, RGSW & GGSW Ciphertexts

Using the GLev ciphertexts introduced in Definition 7, we can define another type of ciphertext, the GGSW ciphertext, that can also be used to describe the public material of some FHE algorithms, in particular the bootstrapping key (Definition 14). Informally, a GGSW ciphertext is a collection of GLev ciphertexts that use the secret key elements to define the GLev plaintexts. These ciphertexts were first introduced in [GSW13].

Definition 8 (GGSW Ciphertexts [GSW13, CLOT21]) *Given a decomposition base $\mathfrak{B} \in \mathbb{N}$ and a decomposition level $\ell \in \mathbb{N}$, a* GGSW ciphertext *of a plaintext $M \in \mathfrak{R}_{q,N}$ under a GLWE secret key $\mathbf{S} = (S_0, \cdots, S_{k-1}) \in \mathfrak{R}_{q,N}^k$ is defined as follows:*

$$\overline{\overline{\mathsf{CT}}} = \big(\overline{\mathsf{CT}}_0, \ldots, \overline{\mathsf{CT}}_k\big) \in \mathsf{GGSW}_{\mathbf{S}}^{\mathfrak{B},\ell}(M) \subseteq \mathfrak{R}_{q,N}^{(k+1)\times\ell\times(k+1)} \tag{4}$$

such that for each $0 \leq i \leq k$,

$$\overline{\mathsf{CT}}_i \in \mathsf{GLev}_{\mathbf{S}}^{\mathfrak{B},\ell}(-S_i \cdot M) \subseteq \mathfrak{R}_{q,N}^{\ell \times (k+1)}.$$

with the convention that $S_k = -1$ by definition.

A GGSW ciphertext with $N = 1$ is called a GSW *ciphertext*, and a GGSW ciphertext with $k = 1$ and $N > 1$ is called a RGSW *ciphertext*.

2.3 Encoding Booleans and Small Integers

In [CGGI20], the authors extensively explained how to use TFHE with Boolean messages. In this section, we explain how to generalize the encoding to support small integers.

Before applying the encryption procedure in Definitions 3 and 4, we first need to encode the message, as explained in the following definition. In TFHE, inputs are most often LWE ciphertexts for which the same blueprint can be used to encode these input messages in the special case where $\mathfrak{R} = \mathbb{Z}$.

Definition 9 (GLWE Encode & Decode) *Let $q \in \mathbb{N}$ be the ciphertext modulus. Let further $p \in \mathbb{N}$ be a* message modulus *and $\pi \in \mathbb{N}$ the number of* bits of padding[2]*, which satisfy $2^\pi \cdot p \leq q$; here $2^\pi \cdot p$ is called the* plaintext modulus*. Let $M \in \mathfrak{R}_{p,N}$ be a message. We define the encoding of M as:*

$$\widetilde{M} = \mathsf{Encode}(M, 2^\pi \cdot p, q) = \lfloor \Delta \cdot M \rceil \in \mathfrak{R}_{q,N} \tag{5}$$

with $\Delta = \frac{q}{2^\pi \cdot p} \in \mathbb{Q}$ referred to as the scaling factor *(see a visual example in Fig. 4 for $N = 1$). To be precise, we lift M to a polynomial with coefficients in $[0, p)$, multiply by Δ, then round each coefficient to an integer and finally reduce modulo q. Note that the final modular reduction will only change q to zero and can only occur when $\pi = 0$. When $\pi > 0$, we may also sometimes want to allow $M \in \mathfrak{R}_{2^\pi \cdot p,N}$ to set explicitly the padding bits, but the encoding procedure remains the same.*

To decode $\widetilde{M} \in \mathfrak{R}_{q,N}$, we compute the following function:

$$M = \mathsf{Decode}\Big(\widetilde{M}, 2^\pi \cdot p, q\Big) = \left\lfloor \frac{\widetilde{M}}{\Delta} \right\rceil \in \mathfrak{R}_{2^\pi \cdot p,N}. \tag{6}$$

To be precise, we lift $\widetilde{M}$ so that it has integer coefficients in $[0, q)$, then divide each coefficient by Δ and round it to the nearest integer and reduce modulo $2^\pi \cdot p$ if necessary.

In practice, we have seen after decryption that $\widetilde{M}$ contains a small error term *$E = \sum_{i=0}^{N-1} e_i \cdot X^i \in \mathfrak{R}$, such that we can write $\widetilde{M} = \Delta \cdot M + E \in \mathfrak{R}_{q,N}$. This error term is the noise in the GLWE ciphertext and thus grows with homomorphic computations. The decoding algorithm fails if and only if there is at least one i, with $0 \leq i < N$ such that $|e_i| \geq \frac{\Delta}{2}$. Letting M be the true underlying message that is the result of performing the computations on unencrypted data, we can write this probability as*

$$\mathsf{Pr}\left(\bigcup_{i=0}^{N-1} |e_i| \geq \frac{\Delta}{2}\right) = \mathsf{Pr}\Big(\mathsf{Decode}\Big(\widetilde{M}, 2^\pi \cdot p, q\Big) \neq M\Big). \tag{7}$$

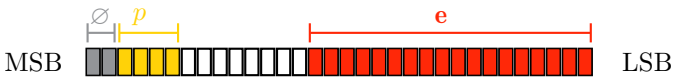

Figure 4: Plaintext binary representation with $\pi = 2$ bits of padding (gray), message modulus $p = 16 = 2^4$ (yellow), such that the plaintext modulus satisfies $2^\pi \cdot p \leq q = 2^{32}$, and the error e (red), the white part is empty. The most significant bits (MSB) are on the left and the least significant bits (LSB) are on the right.

Remark 9 (Why use bits of padding?) *Definition 9 provides the option of using bits of padding in an encoding. At first glance, this may appear like we are wasting space which could be used to*

[2]For simplicity we use a power of 2 for the padding, but this is not a necessary condition.

accommodate larger messages by increasing the message modulus instead. The main reason for this is the PBS operation, which, without a padding bit, can only be used with negacyclic functions. These functions exhibit a form of two-fold symmetry[3]*, which is quite restrictive. Having at least one bit of padding ($\pi \geq 1$) removes this restriction and means that we can apply any function that can be represented using a lookup table during the PBS operation as long as the value of the padding bit is known. Typically, we ensure that the padding bits remain zero throughout. For most applications, $\pi = 1$ is the most suitable choice, however, we will see that encodings using $\pi = 0$ and even $\pi > 1$ have their place in TFHE.*

2.3.1 Carry Space

The encoding introduced in Definition 9 can be modified in order to include an extra *carry space* on the left of the message space – essentially splitting the original message space into a carry subspace and a new, reduced message subspace. We denote by carry the carry modulus and by msg the new message modulus (i.e., $\mathsf{carry} \cdot \mathsf{msg} = p$); cf. Fig. 5.

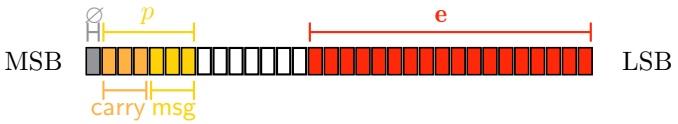

Figure 5: Plaintext binary representation with one bit of padding (gray), carry modulus $\mathsf{carry} = 8 = 2^3$ (orange), the new, reduced message modulus $\mathsf{msg} = 8 = 2^3$ (yellow), so the carry-message modulus is $p = 64 = 2^{3+3}$ (orange+yellow).

As previously mentioned, for the correctness of the PBS, we need the padding bit(s) to be equal to zero (at least to be known). So if we perform addition(s), we are at risk of overwriting the padding bit and thus failing the following bootstrapping. To mitigate this risk, the carry space, originally initialized to zero, is introduced as a dedicated space to absorb the additional bits after performing linear operations, i.e., addition or multiplication by a known integer.

Once the carry space is potentially full, we need to perform a PBS which ensures that the message is reduced modulo msg (i.e., the carry space is cleaned), which in turn provides space for future additions. We remark that this carry-message encoding is one of the cornerstones of the arithmetic operations over large integers that will be introduced later in Section 3.

To give an example, let us assume that we have $\mathsf{msg} = 2^2$ with no carry space (i.e., for now, we assume $\mathsf{carry} = 2^0$), and two messages $m_0 = 3$ and $m_1 = 2$. Both can be written on 2 bits as $m_0 = 11_2$ and $m_1 = 10_2$. However, the sum of m_0 and m_1 can not be written on 2 bits since $m_0 + m_1 = 101_2 = 5$, which motivates the need for a non-trivial carry space.

Another advantage of the carry subspace is that we can delay having to perform a bootstrapping when computing linear operations. This is useful as the PBS operation is much more costly to perform than linear operations.

2.3.2 Boolean Encoding

In TFHE-rs, Boolean values, represented by 1 for true and 0 for false, are encoded within the same carry-message space as small integers (i.e., using the same parameters in practice) with only a simple change: the message part is decreased to a single bit while expanding the carry part to maintain the

[3] See Remark 24 for a precise definition.

overall carry-message size. This means that we have:

$$\mathsf{msg}^{(\mathsf{Bool})} = 2,$$
$$\mathsf{carry}^{(\mathsf{Bool})} = \frac{\mathsf{msg} \cdot \mathsf{carry}}{\mathsf{msg}^{(\mathsf{Bool})}} = \frac{p}{2}.$$

The same rule holds for the carry initialization, i.e., the initial value is zero.

Remark 10 (Boolean Encoding in [CGGI20]) *In [CGGI20], TFHE is described with a different Boolean encoding which has* $\mathsf{msg} = 2$*,* $\mathsf{carry} = 2$*, and* $\pi = 0$*. Thanks to the carry bit, linear combinations between two ciphertexts can be performed before applying a PBS. In this manner, the encoding supports every commutative logical gate with two inputs by choosing the appropriate function to apply during the PBS. Furthermore, the functions evaluated during the PBS can be chosen to be negacyclic (while adding a public constant afterwards to get back to the proper encoding) so we do not require a bit of padding.*

2.3.3 Degree of Fullness

In order to keep track of the worst-case message in a ciphertext – i.e., check if there is still room to perform more linear operations – we implement a piece of metadata that we call the *degree of fullness*.

Definition 10 (Degree of Fullness) *Let* ct *be an* LWE *ciphertext with* π *bits of padding and message (or carry-message) modulus* p*. Let* ct *encrypt a (properly encoded) message* m*,* $0 \leq m \leq \mu < p$*, where* μ *is a publicly known upper bound on* m*. We refer to* μ *as the* degree of fullness*, denoted* $\deg(\mathsf{ct}) \in [0, p-1]$*.*

Remark 11 *Typically, with the carry-message encoding,* $\mathsf{msg} - 1$ *is used as the initial degree of fullness of a fresh ciphertext, unless we have other information. It must be ensured that when entering bootstrapping, the degree of fullness is in the prescribed interval* $[0, p-1]$

In summary, *the carry subspace acts as a buffer* to contain the carry information derived from linear operations, and *the degree of fullness acts as a measure* that indicates when this buffer cannot support any more linear operations. Once this limit is reached, the carry subspace needs to be emptied by applying a PBS.

Remark 12 (Noise Growth) *Generally with FHE, one has to monitor the noise growth in order to guarantee the correctness of successive operations. However, with the concept of a carry buffer, we can choose parameters such that the noise is always under a certain level if the degree of fullness has not reached the maximal value allowed. When we approach this value for the degree, a bootstrapping operation is performed and the noise is reduced at the same time.*

In the rest of this section, we will abstract away the encoding as this allows us to define TFHE operations intuitively as performing some corresponding operation on the encoded message without having to specify exactly which encoding is being used each time. Whether the resulting encoding is sufficient to be able to recover the correct data using some decoding procedure is not discussed here and is much more technically involved. Sufficed to say, correct parameter choices are the key to functional correctness.

2.4 Simple Operations

In this section, we recall the simplest operations that can be performed over ciphertexts, namely addition and multiplication by a public polynomial (scalar multiplication). Then we introduce the sample extraction procedure which converts a GLWE ciphertext into an LWE ciphertext and show how this can be used to construct an efficient public key encryption scheme.

2.4.1 Linear Operations

The first and simplest operation that can be done over ciphertexts is addition. Informally, the addition of two ciphertexts is done by performing coordinate-wise addition modulo q. The resulting noise will be the sum of the two noises. Formally, Algorithm 3 defines how to add two ciphertexts that are encrypted under the same secret key.

Algorithm 3: $\mathsf{CT}_3 \leftarrow \mathsf{Add}(\mathsf{CT}_1, \mathsf{CT}_2)$

Context: $\begin{cases} \mathbf{S} \in \mathfrak{R}_{q,N}^k : \text{the input GLWE secret key} \\ \mathsf{CT}_i = (A_{i,0}, \cdots, A_{i,k-1}, B_i) \in \mathfrak{R}_{q,N}^{k+1} \text{ for } i \in \{1,2\} \end{cases}$

Input: $\begin{cases} \mathsf{CT}_1 \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M_1}\right), \text{ with } \widetilde{M_1} \in \mathfrak{R}_{q,N} \\ \mathsf{CT}_2 \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M_2}\right), \text{ with } \widetilde{M_2} \in \mathfrak{R}_{q,N} \end{cases}$

Output: $\mathsf{CT}_3 \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M_1} + \widetilde{M_2}\right)$

for $0 \leq j \leq k-1$ **do**
  $A_{3,j} \leftarrow A_{1,j} + A_{2,j} \bmod q$
$B_3 \leftarrow B_1 + B_2 \bmod q$
$\mathsf{CT}_3 \leftarrow (A_{3,0}, \cdots, A_{3,k-1}, B_3)$
return CT_3

Theorem 1 (GLWE addition) *Let $\mathbf{S} = (S_0, \cdots, S_{k-1})$ be a GLWE secret key. Suppose that $\mathsf{CT}_1 = (A_{1,0}, \cdots, A_{1,k-1}, B_1)$ and $\mathsf{CT}_2 = (A_{2,0}, \cdots, A_{2,k-1}, B_2)$ are GLWE encryptions of encoded messages $\widetilde{M_1}$ and $\widetilde{M_2}$, respectively, encrypted under the GLWE secret key $\mathbf{S}$ and whose noise polynomials are E_1 and E_2, respectively.*

The addition of CT_1 and CT_2 defined in Algorithm 3 gives a GLWE ciphertext CT_3 which is an encryption of the encoded message $\widetilde{M_1} + \widetilde{M_2}$ under $\mathbf{S}$. Furthermore, the noise in CT_3 is equal to $E_1 + E_2$.

If E_1 and E_2 can be viewed as having been sampled from two centered Gaussian distributions $\chi_{\sigma_1} = \mathcal{N}(0, \sigma_1^2)$ and $\chi_{\sigma_2} = \mathcal{N}(0, \sigma_2^2)$ where $(\sigma_1, \sigma_2) \in \mathbb{R}_{>0}^2$ are the two standard deviations, then under the assumption that χ_{σ_1} and χ_{σ_2} are statistically independent, $E_1 + E_2$ follows a centered Gaussian distribution $\chi_{\sqrt{\sigma_1^2+\sigma_2^2}} = \mathcal{N}(0, \sigma_1^2 + \sigma_2^2)$.

The algoritmic complexity of Algorithm 3 is $\mathbb{C}^{\mathsf{Add}} = (k+1)N\mathbb{C}(\mathsf{add})$, where $\mathbb{C}(\mathsf{add})$ denotes the complexity of adding two elements from $\mathbb{Z}_q$.

Algorithm 3 shows how to add two GLWE ciphertexts, it is easy to generalize this to adding many ciphertexts together. We will also write the addition of ciphertexts using the usual Σ-notation, thus the sum of the GLWE ciphertexts CT_i is written $\sum_i \mathsf{CT}_i$.

With the help of Theorem 1, we can observe that after the addition, the variance of the noise is larger than the noise variances of the input ciphertexts. As previously mentioned, this is not only valid for the addition operation, most operations performed over ciphertexts will increase the variance of the noise. The exceptions being the rotation operation we will see next and of course, the various types of bootstrapping operations we will see later.

Remark 13 (Addition of a public constant) *By having one of the input ciphertexts, say CT_2, in Algorithm 3 be trivial encryption of a public constant that has been suitably encoded, we can add a public constant to a GLWE ciphertext without increasing the noise.*

Before explaining how to multiply a GLWE ciphertext with a polynomial, let us see what happens when we multiply a GLWE ciphertext with a power of X. We call this operation a *rotation*.

Algorithm 4: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{Rotate}(\mathsf{CT}_{\mathsf{in}}, \omega)$

Context: $\begin{cases} \mathbf{S} \in \mathfrak{R}_{q,N}^{k} : \text{the input GLWE secret key} \\ \mathsf{CT}_{\mathsf{in}} = (A_0, \cdots, A_{k-1}, B) \in \mathfrak{R}_{q,N}^{k+1} \end{cases}$

Input: $\begin{cases} \mathsf{CT}_{\mathsf{in}} \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M}\right), \text{ with } \widetilde{M} \in \mathfrak{R}_{q,N} \\ \omega : \text{ an integer} \end{cases}$

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathsf{GLWE}_{\mathbf{S}}\left(X^{\omega} \cdot \widetilde{M}\right)$

for $0 \leq i \leq k-1$ **do**
  $A'_i \leftarrow X^{\omega} \cdot A_i$
$B' \leftarrow X^{\omega} \cdot B$
$\mathsf{CT}_{\mathsf{out}} \leftarrow \left(A'_0, \cdots, A'_{k-1}, B'\right)$
return $\mathsf{CT}_{\mathsf{out}}$

Theorem 2 (Multiplication between a GLWE Ciphertext and a Power of X) *Suppose that $\mathbf{S} = (S_0, \cdots, S_{k-1})$ is a GLWE secret key. Let $\mathsf{CT}_{\mathsf{in}}$ be a GLWE ciphertext encrypting the encoded message $\widetilde{M}$ under the GLWE secret key $\mathbf{S}$ with error polynomial E whose coefficients can be viewed as having been sampled from a symmetric error distribution χ. Here symmetric means that $\mathbb{P}(x \hookleftarrow \chi) = \mathbb{P}(-x \hookleftarrow \chi)$. Also, let ω be an integer.*

The rotation of $\mathsf{CT}_{\mathsf{in}}$ by ω as defined in Algorithm 4 gives a GLWE ciphertext $\mathsf{CT}_{\mathsf{out}}$ which is an encryption under $\mathbf{S}$ of $X^{\omega} \cdot \widetilde{M}$. The noise in $\mathsf{CT}_{\mathsf{out}}$ is equal to $X^{\omega} \cdot E$. Further, each coefficient of the noise in the GLWE ciphertext $\mathsf{CT}_{\mathsf{out}}$ also follows the noise distribution χ.

Note that the rotation of $\mathsf{CT}_{\mathsf{in}}$ depends only on the value of ω modulo $2N$ since $X^{2N} = 1$ in $\mathfrak{R}_{q,N}$.

We remark that both the discrete Gaussian distributions which we use (i.e., those with zero mean), and the TUniform distribution are symmetric.

As we can add GLWE ciphertexts together (Theorem 1) and multiply them by powers of X (Theorem 2), we can combine them to perform the product between a GLWE ciphertext and a polynomial with integer coefficients. We call this operation scalar multiplication, with the integer polynomial being the scalar.

Algorithm 5: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathrm{ScalarMult}(\mathsf{CT}_{\mathsf{in}}, D)$

Context: $\begin{cases} \mathbf{S} \in \mathfrak{R}_{q,N}^{k} : \text{the input GLWE secret key} \\ \mathsf{CT}_{\mathsf{in}} = (A_0, \cdots, A_{k-1}, B) \in \mathfrak{R}_{q,N}^{k+1} \end{cases}$

Input: $\begin{cases} \mathsf{CT}_{\mathsf{in}} \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M}\right), \text{ with } \widetilde{M} \in \mathfrak{R}_{q,N} \\ D \in \mathbb{Z}[X] \end{cases}$

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathsf{GLWE}_{\mathbf{S}}\left(D \cdot \widetilde{M}\right)$

for $0 \leq i \leq k-1$ **do**
  $A'_i \leftarrow D \cdot A_i$
$B' \leftarrow D \cdot B$
$\mathsf{CT}_{\mathsf{out}} \leftarrow \left(A'_0, \cdots, A'_{k-1}, B'\right)$
return $\mathsf{CT}_{\mathsf{out}}$

Theorem 3 (Multiplication between a GLWE Ciphertext and a Polynomial) *Suppose that $\mathbf{S} = (S_0, \cdots, S_{k-1})$ is a GLWE secret key. Let $\mathsf{CT}_{\mathsf{in}}$ be a GLWE ciphertext encrypting the encoded message $\widetilde{M}$ under the GLWE secret key $\mathbf{S}$ with noise polynomial E. Let $D \in \mathbb{Z}[X]$ be such that*

$D = \sum_{i=0}^{N-1} d_i \cdot X^i$. *The scalar multiplication of* $\mathsf{CT}_{\mathsf{in}}$ *with* D *as defined by Algorithm 5 gives a GLWE ciphertext* $\mathsf{CT}_{\mathsf{out}}$ *which is an encryption of* $D \cdot \widetilde{M}$ *under* $\mathbf{S}$. *The noise in* $\mathsf{CT}_{\mathsf{out}}$ *is equal to* $D \cdot E$.

Furthermore, if the coefficients of E *can be viewed as independently sampled from a centered Gaussian distribution* χ_σ, *then each coefficient of the noise in* $\mathsf{CT}_{\mathsf{out}}$ *follows a centered Gaussian distribution* $\chi_{\nu\cdot\sigma}$ *where* ν *is the 2-norm of* D *defined via* $\nu^2 = \sum_{i=0}^{N-1} d_i^2$.

Remark 14 (Fast Polynomial Multiplication) *Naïvely, polynomial multiplication (Lines 2 and 3 of Algorithm 5) needs* $O(N^2)$ *operations. In the TFHE-rs library, polynomial multiplication is implemented with the use of* Fast Fourier Transform *(FFT) which reduces the complexity to* $O(N \log N)$. *However, it comes with a particular caveat: due to the limited precision of floating point types, a new source of error needs to be accounted for.*

Besides FFT-backed fast polynomial multiplication, there exist other methods, in particular, one backed by the Number Theoretic Transform *(NTT), possibly combined with the* Chinese Remainder Theorem *(CRT). This one however requires* q *of a special form: namely a product of more than one prime with some additional properties. Hence, additional error terms due to rounding would be present.*

Note that scalar multiplication is only practical when the scalar is "small" since the noise grows quickly with the 2-norm of the scalar. We will see later how we can get around this requirement that scalars must be small using the idea of decomposition.

Remark 15 (Linear Operations on GLev and GGSW Ciphertexts) *Since both GLev and GGSW ciphertexts can be seen as collections of GLWE ciphertexts, Theorems 1 and 3 can be trivially extended to GLev and GGSW ciphertexts in a component-wise manner.*

Using Theorem 1 and Theorem 3, we can define the TFHE dot product. We present it for GLWE ciphertexts for genericity however in the TFHE-rs library this is mainly used for LWE ciphertexts.

Algorithm 6: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{DotProduct}(\{\mathsf{CT}_i\}_i, \{\omega_i\}_i)$

Context: $\left\{\mathbf{S} = (S_0, S_1, \cdots, S_{k-1}) \in \mathfrak{R}_{q,N}^k : \text{the input GLWE secret key}\right.$

Input: $\begin{cases} \mathsf{CT}_i \in \mathsf{GLWE}_{\mathbf{S}}\left(\sum_{j=0}^{N-1} \widetilde{m_{i,j}} X^j\right), \text{ with } \widetilde{m_{i,j}} \in \mathbb{Z}_q, \text{ for } 0 \leq i < \alpha, 0 \leq j < N \\ \omega_i \in \mathbb{Z}[X] \text{ for } 0 \leq i < \alpha \end{cases}$

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathsf{LWE}_{\mathbf{s}}\left(\widetilde{M_{\mathsf{out}}}\right)$, with $\widetilde{M_{\mathsf{out}}} = \sum_{i=0}^{\alpha-1} \sum_{j=0}^{N-1} \widetilde{m_{i,j}} X^j \cdot \omega_i$

1 $\mathsf{CT}_{\mathsf{out}} \leftarrow \sum_{i=0}^{\alpha-1} \mathsf{ScalarMult}(\mathsf{CT}_i, \omega_i)$
2 **return** $\mathsf{CT}_{\mathsf{out}}$

Theorem 4 (Homomorphic Dot Product) *Let* $\mathsf{ct}_i \in \mathsf{LWE}_{\mathbf{s}}(\widetilde{m_i}) \subseteq \mathbb{Z}_q^{n+1}$ *for* $0 \leq i \leq \alpha - 1$ *be a list of LWE ciphertexts encrypted under an LWE secret key* $\mathbf{s} = (s_0, \cdots, s_{n-1})$ *with noises* e_i *that can be viewed as having been sampled independently from a centered Gaussian distribution* χ_σ. *Let* $\{\omega_i\}_{0 \leq i \leq \alpha-1} \in \mathbb{Z}^\alpha$ *be arbitrary integers which we call weights.*

A homomorphic dot product is a dot product between a vector of LWE ciphertexts and a vector of integers and can be computed using Algorithm 6. It results in an LWE ciphertext $\mathsf{ct}_{\mathsf{out}}$ *encrypting the message* $\sum_{i=0}^{\alpha-1} \widetilde{m_i} \cdot \omega_i$ *under the secret key* $\mathbf{s}$. *The noise in the output ciphertext follows the distribution* $\chi_{\nu\cdot\sigma} = \mathcal{N}\left(0, \nu^2\sigma^2\right)$ *with* $\nu^2 = \sum_{i=0}^{\alpha-1} \omega_i^2$, *the squared 2-norm of the vector of weights.*

Thus, given a dot product between a vector of ciphertexts, whose noises are independently sampled from the same (Gaussian) distribution, and a vector of integers, the output noise of the dot product can be fully characterized by its 2-norm ν.

We will also use the notation $\langle \mathbf{x}, \mathbf{y} \rangle$ to denote the dot product between a vector $\mathbf{x}$ of ciphertexts and a vector $\mathbf{y}$ of scalars. As seen previously, these ciphertexts can be GLWE ciphertexts not just LWE ciphertexts and we will soon see examples of this where the vector of GLWE ciphertexts is a GLev ciphertext.

Remark 16 (Subtraction) *We note that subtraction is a special case of the dot product with $\alpha = 2$, $\omega_0 = 1$ and $\omega_1 = -1$. We will write $\mathsf{CT}_1 - \mathsf{CT}_2$ for the subtraction of CT_2 from CT_1.*

2.4.2 Sample Extract

Algorithm 7: $\mathsf{ct} \leftarrow \mathsf{SE}(\mathsf{CT}, \alpha)$

Context:
$$\begin{cases} \mathbf{s} = (s_0, \cdots, s_{kN-1}) : \text{ the output LWE secret key} \\ \mathbf{S} = (S_0, \ldots, S_{k-1}) : \text{ the input GLWE secret key} \\ \quad \text{where } S_i = \sum_{j=0}^{N-1} s_{i \cdot N + j} X^j, \text{ for } 0 \leq i \leq k-1, \\ \widetilde{M} = \sum_{i=0}^{N-1} \widetilde{m_i} \cdot X^i : \text{ an encoded polynomial message} \\ B = \sum_{j=0}^{N-1} b_j X^j \text{ and } A_i = \sum_{j=0}^{N-1} a_{i,j} X^j \text{ for } 0 \leq i \leq k-1 \end{cases}$$

Input:
$$\begin{cases} \mathsf{CT} = (A_0, \cdots, A_{k-1}, B) \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M}\right) \\ \alpha \in \{0, \cdots, N-1\} \end{cases}$$

Output: $\mathsf{ct} \in \mathsf{LWE}_{\mathbf{s}}(\widetilde{m_\alpha})$

```
b' ← b_α
for 0 ≤ i ≤ k − 1 do
    for 0 ≤ j ≤ α do
        a'_{i·N+j} ← a_{i,α−j}
    for α + 1 ≤ j < N do
        a'_{i·N+j} ← −a_{i,N+α−j}
ct ← (a'_0, ··· , a'_{kN−1}, b')
return ct
```

Sample extract is an operation taking as input a GLWE ciphertext, $\mathsf{CT} \in \mathsf{GLWE}_{\mathbf{S}}\left(\sum_{i=0}^{N-1} \widetilde{m_i} X^i\right)$, and outputting an LWE ciphertext, $\mathsf{ct} \in \mathsf{LWE}_{\bar{\mathbf{s}}}(\widetilde{m_\alpha})$, where $0 \leq \alpha \leq N-1$ is given as a further input (the default being $\alpha = 0$). The output ciphertext ct will be encrypted under a flattened representation of the input GLWE key $\mathbf{S}$. We explain in Definition 11 what a flattened GLWE secret key is.

Definition 11 (Flattened Representation of a GLWE Secret Key) *A GLWE secret key*

$$\mathbf{S} = \left(S_0 = \sum_{j=0}^{N-1} s_{0,j} X^j, \cdots, S_{k-1} = \sum_{j=0}^{N-1} s_{k-1,j} X^j \right) \in \mathfrak{R}_{q,N}^k$$

can be flattened *into an LWE secret key $\bar{\mathbf{s}} = (\bar{s}_0, \cdots, \bar{s}_{kN-1}) \in \mathbb{Z}_q^{kN}$ by defining $\bar{s}_{iN+j} \coloneqq s_{i,j}$, for $0 \leq i < k$ and $0 \leq j < N$.*

A version of the sample extract for RLWE ciphertexts was introduced in [CGGI20, Section 4.2] and can be easily extended to GLWE ciphertexts as is done in Algorithm 7. Informally, the sample extract consists in simply rearranging, possibly with negation, some of the coefficients of the GLWE input ciphertext to build the output LWE ciphertext encrypting one of the coefficients of the input encoded polynomial message.

Theorem 5 (Sample Extract) *Let* $0 \leq \alpha \leq N-1$ *be an index. Let* $\mathsf{CT} \in \mathsf{GLWE}_{\mathbf{S}}\left(\sum_{i=0}^{N-1} \widetilde{m_i} X^i\right)$ *be a GLWE ciphertext with noise polynomial* $E = \sum_{i=0}^{N-1} e_i X^i$ *where* e_i *can be viewed as having been sampled from* χ_σ*. After the sample extract operation, Algorithm 7, we obtain a ciphertext* $\mathsf{ct} \in \mathsf{LWE}_{\bar{\mathbf{s}}}(\widetilde{m_\alpha})$ *encrypted with* $\bar{\mathbf{s}}$*, the flattened representation of* $\mathbf{S}$ *(Definition 11) and the noise in* ct *is* e_α*. As such the noise in the output LWE can also be viewed as being sampled from* χ_σ*, hence sample extract does not add any noise.*

The complexity of the sample extract is:

$$\mathbb{C}^{\mathsf{SE}} = (k+1) \cdot N \cdot \mathbb{C}(\mathsf{copy}) \tag{8}$$

where $\mathbb{C}(\mathsf{copy})$ *denotes the complexity of performing a copy operation on an element in* $\mathbb{Z}_q$*. In practice, most of the time we can neglect the cost of the sample extract as it is very fast compared to the other FHE algorithms.*

2.4.3 Public Key Encryption

As mentioned previously, there are two types of encryption schemes we can consider: a public key scheme and a private key scheme. One can typically go from a private key scheme to a public one by making public a number of encryptions of zero. Here we explain a way to construct a public key scheme for TFHE which has a smaller public key size, based on the work in [Joy24].

The public key consists of a single RLWE sample, $\mathsf{PK} = (A, B) \in \mathsf{RLWE}_S(0)$, so that $B = A \cdot S + E$ for some error term $E \in \mathfrak{R}_{q,N}$ and secret key $S \in \mathfrak{R}_{q,N}$.

Then to encrypt an encoded message $\widetilde{m} \in \mathbb{Z}_q$ using this public key one first samples an element $R \in \mathfrak{R}_{q,N}$ with binary coefficients, two small noise terms $E' \in \mathfrak{R}_{q,N}$ and $e \in \mathbb{Z}_q$ and computes $A' = A \cdot R + E'$, $B' = B \cdot R + e + \widetilde{m}$ (note we add elements of $\mathbb{Z}_q$ to elements of $\mathfrak{R}_{q,N}$ by treating them as constant polynomials). Finally, one applies the sample extraction procedure to the RLWE ciphertext (A', B') (with $\alpha = 0$) to get an LWE ciphertext in $\mathbb{Z}_q^{N+1}$ encrypting $\widetilde{m}$ under the secret key $\mathbf{s}$ that is the flattening of S. A detailed algorithm is given in Algorithm 8.

One can then perform an LWE keyswitch if one wishes to use an LWE dimension that is not N (which must be a power of two).

Algorithm 8: $\mathsf{ct} \leftarrow \mathsf{Encrypt}(\widetilde{m}, \mathsf{PK}; \chi_\sigma)$

Context:
- χ_σ : a noise distribution with standard deviation σ which depends on N
- $\mathcal{U}(\mathfrak{R}_{2,N})$: the uniform distribution over $\mathfrak{R}_{2,N}$, the samples from which we consider in $\mathfrak{R}_{q,N}$
- $\mathbf{s}$: the flatten GLWE secret key S

Input:
- $\mathsf{PK} = (A, B) \in \mathsf{RLWE}_S(0)$: the public key
- $\widetilde{m} \in \mathbb{Z}_q$: an encoded message

Output: $\mathsf{ct} \in \mathsf{LWE}_{\mathbf{s}}(\widetilde{m}) \subseteq \mathbb{Z}_q^{N+1}$

```
for 0 ≤ j ≤ N do
    e_j ↩ χ_σ;
E' ← Σ_{j=0}^{N-1} e_j · X^j;
R ↩ U(R_{2,N});
A' ← A · R + E';
B' ← B · R + e_N + m̃;
ct ← SE((A', B'), 0);
return ct
```

Remark 17 (Zero-Knowledge Proofs) *When using a public key encryption scheme, anyone can encrypt data and send this ciphertext to be computed on homomorphically. This opens up the possibility*

of malicious users sending malformed ciphertexts which could result in information leakage during computation or even degradation of security if not managed appropriately.

To this end, the use of zero-knowledge proofs is employed to give some guarantee that a given ciphertext is not malformed. Put simply, a zero-knowledge proof allows one entity to prove to another that it knows some data without leaking any information about what value the data actually takes. Ideally, in our situation, any ciphertext encrypted using the public key would come with a zero-knowledge proof that the sender knows both the message and the randomness used in generating the ciphertext. In practice, proving only that the randomness generated is smaller than some bound using some norm is enough (the randomness is directly linked to the size of the noise in the output ciphertext). TFHE-rs then provides the ability to verify these zero-knowledge proofs. Details on this can be found in [Lib24].

2.5 Key Switching

We saw with Theorem 1 that we can add ciphertexts encrypted under the same GLWE keys. However, we cannot do the same for ciphertexts encrypted under different keys, even if they have the same dimension. In this subsection, we introduce the concept of a key switch which can solve this issue and more.

A key switch allows to homomorphically change the key a ciphertext is encrypted under. Given a ciphertext $\mathsf{CT}_{\mathsf{in}} \in \mathsf{GLWE}_{\mathbf{S}_{\mathsf{in}}}\left(\widetilde{M}\right)$ we can use a key switch to obtain a ciphertext $\mathsf{CT}_{\mathsf{out}} \in \mathsf{GLWE}_{\mathbf{S}_{\mathsf{out}}}\left(\widetilde{M}\right)$. $\mathbf{S}_{\mathsf{out}}$ can have different parameters than $\mathbf{S}_{\mathsf{in}}$, for instance the polynomial size N and the GLWE dimension k could be different. Several algorithms have been presented in the literature that allow to perform a key switch. Many of them offer some further functionality than simply changing the secret key. We will describe the main versions here. Every version needs some public material to perform the key switch, namely the keyswitching key. Intuitively, the keyswitching key is composed of the key $\mathbf{S}_{\mathsf{in}}$ encrypted under $\mathbf{S}_{\mathsf{out}}$.

2.5.1 Decomposition

First, we introduce the radix decomposition: this algorithm is used as a sub-routine in every key switch algorithm and in other FHE algorithms as well.

Balanced Radix Decomposition. Several ways exist to perform a radix decomposition. The classical radix decomposition was described in [CGGI20]. Here we use a balanced decomposition algorithm inspired by Joye [Joy21] as it ensures that the distribution of the digits of the decomposition has zero mean. We also assume here that $\mathfrak{B}^\ell$ divides q. This assumption can be removed in a number of ways but leads to a more complicated calculation of the resulting noise.

Definition 12 *Let $\mathfrak{B} \in \mathbb{N}$ be a decomposition base and $\ell \in \mathbb{N}$ be a decomposition level such that $\mathfrak{B}^\ell$ divides q. A decomposition of $x \in \mathbb{Z}_q$ with respect to $\mathfrak{B}$ and ℓ is a vector $(x_1, \cdots, x_\ell) \in \mathbb{Z}^\ell$ such that*

$$\left\langle (x_1, \cdots, x_\ell), \left(\frac{q}{\mathfrak{B}} \cdots \frac{q}{\mathfrak{B}^\ell}\right)\right\rangle = \sum_{i=1}^{\ell} x_i \cdot \frac{q}{\mathfrak{B}^i} = \left\lfloor x \cdot \frac{\mathfrak{B}^\ell}{q} \right\rceil \cdot \frac{q}{\mathfrak{B}^\ell} \in \mathbb{Z}_q. \tag{9}$$

We refer to $(x_1, \cdots, x_\ell)$ as the decomposition vector of x or simply the decomposition of x.

In [CGGI20], $(x_1, \cdots, x_\ell)$ are defined as the unique integers satisfying Equation (9) with $-\frac{\mathfrak{B}}{2} \leq x_i < \frac{\mathfrak{B}}{2}$.

In [Joy21], Joye introduces a balanced decomposition algorithm when $q = \mathfrak{B}^\ell$ such that $-\frac{\mathfrak{B}}{2} \leq x_i \leq \frac{\mathfrak{B}}{2}$, which is better for the noise propagation as it gives a centered distribution for the x_i. We note that their algorithm is not deterministic for some inputs. We present a modified version of this

in Algorithm 9 which is deterministic and still has the balanced property we want when $\mathfrak{B}^\ell < q$. We denote this balanced decomposition of x by $\mathsf{BalDec}^{(\mathfrak{B},\ell)}(x)$.

Algorithm 9: $(x_1, \cdots, x_\ell) \leftarrow \mathsf{BalDec}^{(\mathfrak{B},\ell)}(x)$

Context: $\mathfrak{B}$, ℓ : decomposition parameters such that $2\mathfrak{B}^\ell$ divides q

Input: x: an integer in $\mathbb{Z}_q$

Output: $(x_1, \cdots, x_\ell) \in \left[-\frac{\mathfrak{B}}{2}, \frac{\mathfrak{B}}{2}\right]^\ell$, such that Equation (9) holds and the expectation of each digit is zero on uniformly random inputs

1 $y \leftarrow \left\lfloor x \cdot \frac{2\mathfrak{B}^\ell}{q} \right\rceil$
2 $\alpha \leftarrow [y]_2$
3 $z \leftarrow \left\lfloor \frac{y+1}{2} \right\rceil$
4 **if** $z > \frac{\mathfrak{B}^\ell}{2}$ **or** $\left(z = \frac{\mathfrak{B}^\ell}{2}$ **and** $\alpha = 1\right)$ **then**
5 $z \leftarrow z - \mathfrak{B}^\ell$
6 **for** $0 \leq i \leq \ell - 1$ (in ascending order) **do**
7 $z_i \leftarrow z \bmod \mathfrak{B}$ // So $0 \leq z_i < \mathfrak{B}$
8 $z \leftarrow (z - z_i)/\mathfrak{B}$
9 **if** $z_i > \frac{\mathfrak{B}}{2}$ **or** $\left(z_i = \frac{\mathfrak{B}}{2}$ **and** $(z \bmod \mathfrak{B}) \geq \frac{\mathfrak{B}}{2}\right)$ **then**
10 $z_i \leftarrow z_i - \mathfrak{B}$
11 $z \leftarrow z + 1$
12 $x_{\ell-i} \leftarrow z_i$
13 **return** $(x_1, \cdots, x_\ell)$

Remark 18 (TFHE-rs specific implementation of BalDec) *Algorithm 9 as implemented in the TFHE-rs library uses a specific world size, which we denote by BITS, and elements of $\mathbb{Z}_q$ are stored in the top $\log_2(q)$ bits of each word, hence the bottom $\mathsf{BITS} - \log_2(q)$ will be zero. This means that q in Line 1 will be replaced by 2^{BITS}. Furthermore, Line 5 is performed modulo 2^{BITS} which can affect the value of $z_{\ell-1}$ if it is initally set to $\mathfrak{B}/2$. In particular, this means that the balanced property is not preserved for all parameter choices and we have the additional restriction that $\mathfrak{B}^{\ell+1}$ must divide 2^{BITS} in order to have a correctly balanced decomposition.*

Decomposition can naturally be generalized to work for a wider range of input types. When applying the decomposition on a vector of integers, the result is a vector of decomposition vectors of integers. It is also possible to decompose an integer polynomial $P \in \mathfrak{R}_q$ with this algorithm in a componentwise manner. The result will satisfy

$$\left\langle \mathsf{BalDec}^{(\mathfrak{B},\ell)}(P), \left(\frac{q}{\mathfrak{B}} \cdots \frac{q}{\mathfrak{B}^\ell}\right) \right\rangle = \left\lfloor P \cdot \frac{\mathfrak{B}^\ell}{q} \right\rceil \cdot \frac{q}{\mathfrak{B}^\ell} \in \mathfrak{R}_{q,N}$$

with $\mathsf{BalDec}^{(\mathfrak{B},\ell)}(P)$ being a vector of polynomials. Applying this kind of decomposition on a vector of polynomials, we get a vector of vectors of polynomials, and so on.

With decomposition now defined, we can see the utility of the GLev ciphertexts at work.

2.5.2 LWE and GLWE Key Switch

Using Theorem 1 and Definition 12, Algorithm 10 defines the LWE key switch as was introduced in [CGGI17, Algorithm 1]. The output ciphertext is a GLWE ciphertext which, of course, can also be an LWE ciphertext when $N = 1$, which is the typical case.

Algorithm 10: $\mathsf{CT} \leftarrow \mathsf{KS}(\mathsf{ct}, \mathsf{KSK})$

Context: $\begin{cases} \mathbf{s} = (s_0, \cdots, s_{n-1}) : \text{ the input LWE secret key} \\ \mathbf{S} = (S_0, \ldots, S_{k-1}) : \text{ the output GLWE secret key} \end{cases}$

Input: $\begin{cases} \mathsf{ct} = (a_0, \cdots, a_{n-1}, b) \in \mathsf{LWE}_{\mathbf{s}}(\widetilde{m}) \\ \mathsf{KSK} = \left\{ \overline{\mathsf{CT}}_i \in \mathsf{GLev}_{\mathbf{S}}^{\mathfrak{B},\ell}(s_i) \right\}_{0 \leq i \leq n-1} : \text{ keyswitching key from } \mathbf{s} \text{ to } \mathbf{S} \end{cases}$

Output: $\mathsf{CT} \in \mathsf{GLWE}_{\mathbf{S}}(\widetilde{m})$

$\mathsf{CT} \leftarrow (0, \cdots, 0, b) - \sum_{i=0}^{n-1} \left\langle \overline{\mathsf{CT}}_i, \mathsf{BalDec}^{(\mathfrak{B},\ell)}(a_i) \right\rangle$

return CT

Theorem 6 (LWE Key Switch) *Let $\mathsf{ct} \in \mathsf{LWE}_{\mathbf{s}}(\widetilde{m}) \subseteq \mathbb{Z}_q^{n+1}$ be an LWE ciphertext encrypting $\widetilde{m} \in \mathbb{Z}_q$, under the LWE secret key $\mathbf{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{Z}_q^n$, with noise which can be viewed as having been sampled from χ_σ. Let $\mathbf{S}$ be a GLWE secret key such that $\mathbf{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N}^k$. Let $\mathsf{KSK} = \left\{ \overline{\mathsf{CT}}_i \in \mathsf{GLev}_{\mathbf{S}}^{\mathfrak{B},\ell}(s_i) \subseteq \mathfrak{R}_{q,N}^{\ell \times (k+1)} \right\}_{0 \leq i \leq n-1}$ be a key switching key from $\mathbf{s}$ to $\mathbf{S}$ with noise sampled from $\chi_{\sigma_{\mathsf{KSK}}}$.*

The variance of the j-th coefficient of the noise in the output of Algorithm 10 is

$$\begin{aligned} \mathsf{Var}_{\mathsf{KS}}[j] = &\left(\sigma^2 + n \cdot \left(\frac{q^2}{12\mathfrak{B}^{2\ell}} + c_{var} \right) \cdot \mathsf{Var}(s_i) + n \cdot \left(\frac{q^2}{12\mathfrak{B}^{2\ell}} + c_{exp} \right) \cdot \mathbb{E}^2(s_i) \right) \cdot \delta_{j,0} \\ &+ n \cdot \ell \cdot \sigma_{\mathsf{KSK}}^2 \cdot \left(\frac{\mathfrak{B}^2}{12} + \varepsilon(\mathfrak{B}, \ell) \right), \end{aligned} \tag{10}$$

where c_{var} and c_{exp} are small constants that depend on exactly how the rounding operation is performed and $\varepsilon(\mathfrak{B}, \ell) = -\frac{1}{12} + \frac{1}{4(\mathfrak{B}+1)} + \frac{1-(-\mathfrak{B})^{-\ell}}{\ell \cdot (1+\mathfrak{B}^{-1})^2} \in \mathcal{O}(1)$ is a small correction coming from the distribution of the digits output by Algorithm 9, other decomposition algorithms would change this value slightly. Here, $\delta_{j,0}$ is equal to 1 if $j = 0$, else it is zero.

The algorithmic complexity of Algorithm 10 is

$$\begin{aligned} \mathbb{C}^{\mathsf{KS}} = &n\mathbb{C}^{(\ell)}(\mathsf{BalDec}) + \ell n(k+1)N\mathbb{C}(\mathsf{mul}) \\ &+ ((\ell n - 1)(k+1)N)\mathbb{C}(\mathsf{add}) . \end{aligned} \tag{11}$$

where $\mathbb{C}(\mathsf{mul})$ denotes the complexity of performing a multiplication between two elements in $\mathbb{Z}_q$, and $\mathbb{C}(\mathsf{BalDec})$ denotes the complexity of the balanced decomposition outlined in Algorithm 9. The size (in bits) of the key switching key $\mathsf{Size}(\mathsf{KSK})$ is computed with the following formula:

$$\mathsf{Size}(\mathsf{KSK}) \coloneqq n \cdot \ell \cdot (k+1) \cdot N \cdot \lceil \log_2(q) \rceil$$

Remark 19 (Stochastic Rouding) *As an example of a particular rounding operation, if one rounds to the nearest integer and if there is a tie (i.e., we are rounding a value that is exactly half way between two integers), then we flip a random bit to determine which of the two closest integers is chosen, then the constants are $c_{var} = c_{exp} = \frac{1}{6}$. This rounding has the nice property that the expectation of the noise coming from the rounding of a uniformly distributed integer in $\mathbb{Z}_q$ is zero. This can also be achieved by deterministic methods of rounding which depend on q.*

Remark 20 (Public functional Key Switch) *It is possible to leverage the key switch algorithm to also compute a public linear function $f : \mathbb{Z}_q^\alpha \to \mathfrak{R}_{q,N}$ on several input LWE ciphertexts $\{\mathsf{ct}_i\}_{i=0}^{\alpha-1}$. In particular, f must satisfy the following properties:*

- *for any* $(x, y) \in \left(\mathbb{Z}_q^\alpha\right)^2$ *we have* $f(x+y) = f(x) + f(y)$*;*
- *for any* $t \in \mathbb{Z}$ *and any* $x \in \mathbb{Z}_q^\alpha$ *we have* $f(t \cdot x) = t \cdot f(x)$*.*

To perform a public functional key switch, one computes

$$\mathsf{CT} \leftarrow (0, \cdots, 0, f((b_0, \cdots, b_{\alpha-1}))) - \sum_{i=0}^{n-1} \left\langle \overline{\mathsf{CT}}_i, \mathsf{BalDec}^{(\mathfrak{B},\ell)}(f((a_{i,0}, \cdots, a_{i,\alpha-1}))) \right\rangle.$$

If ct_i *encrypt the encoded message* $\widetilde{m_i}$ *then* CT *will encrypt* $f(\widetilde{m_0}, \cdots, \widetilde{m_{\alpha-1}})$*. It is given in algorithmic form in Algorithm 11. This operation was first presented in [CGGI17, Algorithm 1] and can be used to pack many LWE ciphertexts into a single GLWE ciphertext, see Section 2.5.3 for details on this.*

Algorithm 11: $\mathsf{CT} \leftarrow \mathsf{PublicKS}(\{\mathsf{ct}_i\}_{i=1}^{\alpha}, \mathsf{KSK}, f)$

Context: $\begin{cases} \mathbf{s} = (s_0, \cdots, s_{n-1}) \in \mathbb{Z}_q^n : \text{ the input LWE secret key} \\ \mathbf{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N}^k : \text{ the output GLWE secret key} \end{cases}$

Input: $\begin{cases} \{\mathsf{ct}_i = (a_{i,0}, \cdots, a_{i,n-1}, b_i) \in \mathsf{LWE}_{\mathbf{s}}(\widetilde{m_i})\}_{i=0}^{\alpha-1} \text{ with } \widetilde{m_i} \in \mathbb{Z}_q \\ \mathsf{KSK} = \left\{\overline{\mathsf{CT}}_i \in \mathsf{GLev}_{\mathbf{S}}^{\mathfrak{B},\ell}(s_i)\right\}_{0 \le i \le n} : \text{ public functional keyswitching key from } \mathbf{s} \text{ to } \mathbf{S} \\ f : \text{ the public linear function } \mathbb{Z}_q^\alpha \to \mathfrak{R}_{q,N} \text{ to be homomorphically evaluated} \end{cases}$

Output: $\mathsf{CT} \in \mathsf{GLWE}_{\mathbf{S}}\left(f\left(\widetilde{m_0}, \cdots, \widetilde{m_{\alpha-1}}\right)\right)$

$\mathsf{CT} \leftarrow (0, \cdots, 0, f((b_0, \cdots, b_{\alpha-1}))) - \sum_{i=0}^{n-1} \left\langle \overline{\mathsf{CT}}_i, \mathsf{BalDec}^{(\mathfrak{B},\ell)}(f((a_{i,0}, \cdots, a_{i,\alpha-1}))) \right\rangle$

return CT

We now specify in Algorithm 12, a GLWE key switch that works the same way as the LWE key switch of Algorithm 10. The dot product between the keyswitching key and the polynomial decompositions can be performed using the FFT which significantly speeds up the execution time of the GLWE key switch compared to the LWE key switch.

Algorithm 12: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{GlweKeySwitch}(\mathsf{CT}_{\mathsf{in}}, \mathsf{KSK})$

Context: $\begin{cases} \mathbf{S}_{\mathsf{in}} = (S_{\mathsf{in},0}, \cdots, S_{\mathsf{in},k_{\mathsf{in}}-1}) \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}} : \text{the input GLWE secret key} \\ \mathbf{S}_{\mathsf{out}} \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}} : \text{the output GLWE secret key} \end{cases}$

Input: $\begin{cases} \mathsf{CT}_{\mathsf{in}} = (A_0, \cdots, A_{k_{\mathsf{in}}-1}, B) \in \mathsf{GLWE}_{\mathbf{S}_{\mathsf{in}}}\left(\widetilde{M}\right) \subseteq \mathfrak{R}_{q,N}^{k_{\mathsf{in}}+1}, \text{ with } \widetilde{M} \in \mathfrak{R}_{q,N} \\ \mathsf{KSK} = \left\{\overline{\mathsf{CT}}_i \in \mathsf{GLev}_{\mathbf{S}_{\mathsf{out}}}^{\mathfrak{B},\ell}(S_{\mathsf{in},i})\right\}_{0 \le i \le k_{\mathsf{in}}-1} : \text{keyswitching key from } \mathbf{S}_{\mathsf{in}} \text{ to } \mathbf{S}_{\mathsf{out}} \end{cases}$

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathsf{GLWE}_{\mathbf{S}_{\mathsf{out}}}\left(\widetilde{M}\right)$

$\mathsf{CT}_{\mathsf{out}} \leftarrow (0, \cdots, 0, B) \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}+1}$

for $0 \le i \le k_{\mathsf{in}} - 1$ **do**

  $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{CT}_{\mathsf{out}} - \left\langle \overline{\mathsf{CT}}_i, \mathsf{BalDec}^{(\mathfrak{B},\ell)}(A_i) \right\rangle$

return $\mathsf{CT}_{\mathsf{out}}$

Theorem 7 (GLWE Key Switch) *Let* $\mathsf{CT} \in \mathsf{GLWE}_{\mathbf{S}_{\mathsf{in}}}\left(\widetilde{M}\right) \subseteq \mathfrak{R}_{q,N}^{k_{\mathsf{in}}+1}$ *be a GLWE ciphertext encrypting* $\widetilde{M} \in \mathfrak{R}_{q,N}$*, under the GLWE secret key* $\mathbf{S}_{\mathsf{in}} = (S_{\mathsf{in},0}, \ldots, S_{\mathsf{in},k_{\mathsf{in}}-1}) \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}}$*, with noise coefficients which can be viewed as having been sampled from* χ_σ*. Let* $\mathbf{S}_{\mathsf{out}} \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}}$ *be another GLWE secret*

key and let $\mathsf{KSK} = \left\{\overline{\mathsf{CT}}_i \in \mathsf{GLev}^{\mathfrak{B},\ell}_{\mathbf{S}_{\mathsf{out}}}(S_{\mathsf{in},i}) \subseteq \mathfrak{R}^{\ell\times(k_{\mathsf{out}}+1)}_{q,N}\right\}_{0\leq i\leq k_{\mathsf{in}}-1}$ *be a GLWE key switching key from* $\mathbf{S}_{\mathsf{in}}$ *to* $\mathbf{S}_{\mathsf{out}}$ *with noise sampled from* $\chi_{\sigma_{\mathsf{KSK}}}$.

The variance of the j*-th coefficient of the noise in the output of Algorithm 10 is*

$$\begin{aligned}\mathsf{Var}_{\mathsf{KS}}[j] = \sigma^2 &+ k\cdot N\cdot\left(\frac{q^2}{12\mathfrak{B}^{2\ell}}+c_{var}\right)\cdot\mathsf{Var}(S_{\mathsf{in},i}) + k\cdot N\cdot\left(\frac{q^2}{12\mathfrak{B}^{2\ell}}+c_{exp}\right)\cdot\mathbb{E}^2(S_{\mathsf{in},i})\\ &+ k\cdot N\cdot\ell\cdot\sigma^2_{\mathsf{KSK}}\cdot\left(\frac{\mathfrak{B}^2}{12}+\varepsilon(\mathfrak{B},\ell)\right),\end{aligned} \tag{12}$$

where c_{var}*,* c_{exp} *and* $\varepsilon(\mathfrak{B},\ell)$ *are defined in Theorem 6, and* $\mathsf{Var}(S_{\mathsf{in},i})$*, respectively* $\mathbb{E}^2(S_{\mathsf{in},i})$*, is the average of the variances, respectively mean squareds, of the coefficients of* $S_{\mathsf{in},i}$.

The algorithmic complexity of Algorithm 10 is

$$\begin{aligned}\mathbb{C}^{\mathsf{KS}} = {} & k_{\mathsf{in}}N\mathbb{C}^{(\ell)}(\mathsf{BalDec}) + \ell k_{\mathsf{in}}(k_{\mathsf{out}}+1)N\log(N)\mathbb{C}(\mathsf{mul})\\ & + ((\ell k_{\mathsf{in}}-1)(k_{\mathsf{out}}+1)N)\mathbb{C}(\mathsf{add})\,.\end{aligned} \tag{13}$$

The size (in bits) of the key switching key $\mathsf{Size}(\mathsf{KSK})$ *is computed with the following formula:*

$$\mathsf{Size}(\mathsf{KSK}) \coloneqq k_{\mathsf{in}}\cdot\ell\cdot(k_{\mathsf{out}}+1)\cdot N\cdot\lceil\log_2(q)\rceil$$

2.5.3 Packing Key Switch

The idea of a packing keyswitch is to pack several LWE ciphertexts into a single GLWE ciphertext. It takes as input a set of $\alpha \leq N$ LWE ciphertexts $\mathsf{ct}_0, \mathsf{ct}_1, \cdots, \mathsf{ct}_{\alpha-1}$ as well as a set of α indices $\{i_j\}_{j=0}^{\alpha-1}$. Given this set of indices and some public material KSK, a packing keyswitch packs the α messages contained within the LWE ciphertexts into a single GLWE ciphertext $\mathsf{GLWE}\left(\sum_{j=0}^{\alpha-1}\widetilde{m_j}\cdot X^{i_j}\right)$ where $\widetilde{m_j}\in\mathbb{Z}_q$ is the encoded message encrypted in ct_j. We present the full algorithm in Algorithm 13.

Algorithm 13: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{PackingKS}(\{\mathsf{ct}_j\}_{j=0}^{\alpha-1}, \{i_j\}_{j=0}^{\alpha-1}, \mathsf{KSK})$

Context: $\begin{cases}\mathbf{s}=(s_0,\cdots,s_{n-1})\in\mathbb{Z}_q^n: \text{ the input LWE secret key}\\ \mathbf{S}=(S_0,\ldots,S_{k-1})\in\mathfrak{R}^k_{q,N}: \text{ the output GLWE secret key}\end{cases}$

Input: $\begin{cases}\{\mathsf{ct}_j=(a_{j,0},\cdots,a_{j,n-1},b_j)\in\mathsf{LWE}_{\mathbf{s}}(\widetilde{m_j})\}_{j=0}^{\alpha-1} \text{ with } \widetilde{m_j}\in\mathbb{Z}_q\\ \mathsf{KSK}=\left\{\overline{\mathsf{CT}}_i\in\mathsf{GLev}^{\mathfrak{B},\ell}_{\mathbf{S}}(s_i)\right\}_{0\leq i\leq n}: \text{ public functional keyswitching key from } \mathbf{s} \text{ to } \mathbf{S}\end{cases}$

Output: $\mathsf{CT}\in\mathsf{GLWE}_{\mathbf{S}}\left(\sum_{j=0}^{\alpha-1}\widetilde{m_j}\cdot X^{i_j}\right)$

$\mathsf{CT}\leftarrow\left(0,\cdots,0,\sum_{j=0}^{\alpha-1}b_j\cdot X^{i_j}\right)-\sum_{i=0}^{n-1}\left\langle\overline{\mathsf{CT}}_i,\mathsf{BalDec}^{(\mathfrak{B},\ell)}\left(\sum_{j=0}^{\alpha-1}a_{j,i}\cdot X^{i_j}\right)\right\rangle$

return CT

We have already seen one method that can be used to perform a packing keyswitch, namely the public functional keyswitch from Remark 20. This packing key switch can be described easily with the help of the LWE key switch (Algorithm 10 and Theorem 6), the multiplication by a power of X (Theorem 2) and the addition (Theorem 1). In short, it takes as input several LWE ciphertexts, keyswitches them using Algorithm 10, multiplies each of them by a power of X, and finally adds the resulting ciphertexts together.

The noise in this case can be derived from Eq. (10). Define $\delta_{j,\mathcal{I}}$, with $\mathcal{I}=\{i_j\}_{j=0}^{\alpha-1}$ the set of indices used, such that $\delta_{j,\mathcal{I}}$ is equal to 1 if $j\in\mathcal{I}$ and zero otherwise. Then the noise of using Algorithm 10 to

perform a packing keyswitch has variance in the jth coefficient of

$$\mathsf{Var}_{\mathbf{PKS}}[j] = \left(\sigma^2 + n \cdot \left(\frac{q^2}{12\mathfrak{B}^{2\ell}} + c_{var}\right) \cdot \mathsf{Var}(s_i) + n \cdot \left(\frac{q^2}{12\mathfrak{B}^{2\ell}} + c_{exp}\right) \cdot \mathbb{E}^2(s_i)\right) \cdot \delta_{j,\mathcal{I}} + \alpha \cdot n \cdot \ell \cdot \sigma^2_{\mathsf{KSK}} \cdot \left(\frac{\mathfrak{B}^2}{12} + \varepsilon(\mathfrak{B}, \ell)\right),$$

The complexity is clearly α times that of Eq. (11).

Remark 21 (Alternative Packing Keyswitches) *In [CDKS21], using automorphisms to compute the trace function from algebraic number theory, the authors introduced a way to pack many LWE ciphertexts into a single RLWE ciphertext (this can be naturally extended to a GLWE ciphertext). While more efficient than the traditional packing key switch, there is less flexibility in the input and output dimensions as the associated keys are related. Another approach that exploits GLWE ciphertexts (called MLWE in the paper) is HERMES [BCK+23].*

2.6 Building Blocks of the PBS

In this section, we define the building blocks of TFHE's PBS as described in [CGGI16b, CGGI17, CGGI20]. We have already seen one of these, namely the sample extract Algorithm 7. Further, we need to introduce the modulus switch and the external product operations. Then, on top of the external product, we build the *controlled mutex* (CMux), and from the CMux, we explain how to perform a blind rotate.

2.6.1 Modulus Switch

The modulus switch takes as input an LWE ciphertext with some ciphertext modulus q and outputs an LWE ciphertext with ciphertext modulus w. If the input ciphertext encrypts the encoded message $\tilde{m}$ then the output ciphertext will encrypt the encoded message $\left\lfloor \tilde{m} \cdot \frac{w}{q} \right\rceil$. In the context of TFHE's PBS, $w = 2 \cdot N$ with N the polynomial size of the ring $\mathfrak{R}_{q,N}$. There are two variants of the modulus switching algorithm, indicated by the boolean value DriftMitigation. If the DriftMitigation flag is set to true, then the input ciphertext is modified in order to reduce the overall noise variance of the output ciphertext. To do so, we need an additional public key called DMK. The modifications to the input ciphertext for DriftMitigation = True are described in Algorithm 15 and the modulus switching algorithm itself is described in Algorithm 14.

Drift Mitigation. As mentioned in Section 2.2.1, decryption produces a noisy version of the encoded message contained in the ciphertext. The failure probability (p_{fail}) represents the likelihood of obtaining an incorrect message after decryption and decoding. As such, it largely depends on the magnitude of the noise in the ciphertext.

The noise of the modulus switching is largely determined by the rounding errors. The vector of all rounding errors that occur during modulus switching is called the drift vector. The additional noise generated by the modulus switching operation arises from the dot product between the drift vector and the secret key. As the size of the rounding errors depends on the mask of the input ciphertext, an encryption of zero can be added to the input ciphertext to modify the mask without affecting the underlying message. Hence by carefully selecting an appropriate encryption of zero, the rounding errors can partially be mitigated, such that even though adding an encryption of zero increases the noise of the input ciphertext of the modulus switching operation, the noise of the output ciphertext will be reduced. More details on this technique can be found in [BJSW24].

Algorithm 14: $\mathsf{ct_{out}} \leftarrow \mathsf{MS}(\mathsf{ct_{in}}, w, \mathsf{DriftMitigation}, \mathsf{DMK})$

Context: $\left\{\mathbf{s} = (s_0, \cdots, s_{n-1}) \in \mathbb{Z}_q^n : \text{ the LWE secret key}\right.$

Input:
$\begin{cases} \mathsf{ct_{in}} = (a_0, \cdots, a_{n-1}, b) \in \mathsf{LWE_s}(\widetilde{m}) \subseteq \mathbb{Z}_q^{n+1} \\ w : \text{ the output ciphertext modulus} \\ \mathsf{DriftMitigation} : \text{ Boolean value indicating if the drift mitigation technique should be applied} \\ \mathsf{DMK} : \text{ drift mitigation key, used when } \mathsf{DriftMitigation} \text{ is True.} \end{cases}$

Output: $\mathsf{ct_{out}} \in \mathsf{LWE_s}\left(\left\lfloor \frac{\widetilde{m} \cdot w}{q} \right\rceil\right) \subseteq \mathbb{Z}_w^{n+1}$

if DriftMitigation **then**
  $\mathsf{ct}' = (a'_0, \cdots, a'_{n-1}, b') \leftarrow \mathsf{DriftMitigation}(\mathsf{ct_{in}}, \mathsf{DMK})$
else
  $\mathsf{ct}' = (a'_0, \cdots, a'_{n-1}, b') \leftarrow \mathsf{ct_{in}}$
for $0 \leq i < n$ **do**
  $a''_i \leftarrow \left[\left\lfloor \frac{a'_i \cdot w}{q} \right\rceil\right]_w$
$b'' \leftarrow \left[\left\lfloor \frac{b' \cdot w}{q} \right\rceil\right]_w$
$\mathsf{ct_{out}} \leftarrow (a''_0, \cdots, a''_{n-1}, b'')$
return $\mathsf{ct_{out}}$

To apply this technique, additional public key material is required, specifically a set of encryptions of zero called $\mathsf{DMK} = \left\{\mathsf{ct}_z \in \mathsf{LWE_s}(0) \subseteq \mathbb{Z}_q^{n+1}\right\}_{z \in [1,Z]}$. Additionally, an extra computation step is needed: an encryption of zero must be added to the LWE ciphertext input before performing modulus switching.

Let T denote the bound on the maximum allowed modulus switch noise (in absolute value) and r the parameter such that the probability of the modulus switch noise lying in the interval $[\tilde{\mu} - \mathsf{r} \cdot \tilde{\sigma}, \tilde{\mu} + \mathsf{r} \cdot \tilde{\sigma}] \subset [-\mathsf{T}, \mathsf{T}]$, with $\tilde{\mu}$ and $\tilde{\sigma}$ computed as in Algorithm 15, equals $1 - \mathsf{erfc}(\mathsf{r}/\sqrt{2})$. The total number of encryptions of zero needed in the public key material is set such that the chance of not finding an encryption of zero that makes the modulus switch noise satisfy the upper bound T is negligible.[4] The usage of the drift mitigation is indicated with a boolean input value DriftMitigation to the modulus switching given in Algorithm 14, which is by default set to true as the cost of adding an encryption of zero is negligible compared to the benefits of the noise reduction it triggers.

Theorem 8 (Modulus Switch) *Let q and w be two ciphertext moduli. Let $\mathbf{s}$ be an LWE secret key such that $\mathbf{s} = (s_0, \cdots, s_{n-1})$. Let $\mathsf{ct_{in}}$ be an LWE ciphertext (Definition 3) such that $\mathsf{ct_{in}} \in \mathsf{LWE_s}(\widetilde{m}) \subseteq \mathbb{Z}_q^{n+1}$, so that it has ciphertext modulus q.*

The output of the modulus switch operation (Algorithm 14) is a ciphertext $\mathsf{ct_{out}}$ with ciphertext modulus w such that $\mathsf{ct_{out}} \in \mathsf{LWE_s}\left(\left\lfloor \tilde{m} \cdot \frac{w}{q} \right\rceil\right) \subseteq \mathbb{Z}_w^{n+1}$.

Let us assume that w divides q and that q/w is even, as is the case for the modulus switch in the PBS and denote the variance of the noise in $\mathsf{ct_{in}}$ by σ^2_{in}. If the boolean DriftMitigation is false, then the variance of the noise in $\mathsf{ct_{out}}$ is:

$$\mathsf{Var}(\mathsf{MS}) = \sigma^2_{\mathsf{in}} \cdot \frac{w^2}{q^2} + \frac{1}{12} + c_{var}\frac{w^2}{q^2} + n\left(\left(\frac{1}{12} + c_{var}\frac{w^2}{q^2}\right)\mathsf{Var}(s_i) + \left(\frac{1}{12} + c_{exp}\frac{w^2}{q^2}\right)\mathbb{E}^2(s_i)\right); \quad (14)$$

otherwise, the boolean DriftMitigation is true and, denoting the variance of the noise of the selected fresh encryption of zero, $\mathsf{ct_{index}}$, by $\sigma^2_{\mathsf{index}}$, then the variance of the noise in $\mathsf{ct_{out}}$ is:

[4] If it happens that no encryption of zero results in a modulus switch noise that is lower than the upper bound, the encryption of zero that is closest to the upper bound is taken. However, in practice, this should never happen as we can choose the probability of not finding a candidate in the list to be negligible compared with the failure probability p_{fail}.

Algorithm 15: $\mathsf{ct_{out}} \leftarrow \mathsf{DriftMitigation}(\mathsf{ct_{in}}, \mathsf{DMK})$

Context: $\begin{cases} \mathsf{s} = (s_0, \cdots, s_{n-1}) \in \mathbb{Z}^n : \text{ the input binary LWE secret key} \\ \mathsf{T},\ \mathsf{r},\ Z : \text{ the drift parameters} \end{cases}$

Input: $\begin{cases} \mathsf{ct_{in}} \in \mathsf{LWE_s}(\widetilde{m}) \subseteq \mathbb{Z}_q^{n+1}, \text{ where } \widetilde{m} = \Delta \cdot m \\ \mathsf{DMK} \text{ the drift mitigation key} \end{cases}$

Output: $\mathsf{ct_{out}} \in \mathsf{LWE_s}(\widetilde{m})$

$\mathsf{index} = 0$
$\mathsf{ct_{in}} = (a_{\mathsf{in},0}, \cdots, a_{\mathsf{in},n-1}, b_{\mathsf{in}})$
$\tilde{\mu} \leftarrow \left(\left\lfloor b_{\mathsf{in}} \frac{2N}{q} \right\rceil \frac{q}{2N} - b_{\mathsf{in}}\right) - \sum_{j=0}^{n-1} \left(\left\lfloor a_{\mathsf{in},j} \frac{2N}{q} \right\rceil \frac{q}{2N} - a_{\mathsf{in},j}\right) \cdot \frac{1}{2}$
$\tilde{\sigma}^2 \leftarrow \sum_{j=0}^{n-1} \left(\left\lfloor a_{\mathsf{in},j} \frac{2N}{q} \right\rceil \frac{q}{2N} - a_{\mathsf{in},j}\right)^2 \cdot \frac{1}{4}$
$\mathsf{T_{best}} = |\tilde{\mu}| + \mathsf{r} \cdot \tilde{\sigma}$
$\mathsf{index} = 0$
while $|\tilde{\mu}| + \mathsf{r} \cdot \tilde{\sigma} \geq \mathsf{T}$ **and** $\mathsf{index} < Z$ **do**
  $\mathsf{index} \leftarrow \mathsf{index} + 1$
  $\mathsf{ct_{DriftMitigation}} = (a_0, \cdots, a_{n-1}, b) \leftarrow \mathbf{Add}(\mathsf{ct_{in}}, \mathsf{ct_{index}})$
  $\tilde{\mu} \leftarrow \left(\left\lfloor b \frac{2N}{q} \right\rceil \frac{q}{2N} - b\right) - \sum_{j=0}^{n-1} \left(\left\lfloor a_j \frac{2N}{q} \right\rceil \frac{q}{2N} - a_j\right) \cdot \frac{1}{2}$
  $\tilde{\sigma}^2 \leftarrow \sum_{j=0}^{n-1} \left(\left\lfloor a_j \frac{2N}{q} \right\rceil \frac{q}{2N} - a_j\right)^2 \cdot \frac{1}{4}$
  $\mathsf{T_{index}} = |\tilde{\mu}| + \mathsf{r} \cdot \tilde{\sigma}$
  if $\mathsf{T_{index}} < \mathsf{T_{best}}$ **then**
    $\mathsf{T_{best}} = \mathsf{T_{index}}$
    $\mathsf{index_{best}} = \mathsf{index}$
if $\mathsf{index} = Z$ **and** $\mathsf{T_{index}} > \mathsf{T}$ **and** $\mathsf{index_{best}} \neq 0$ **then**
  $\mathsf{ct_{DriftMitigation}} \leftarrow \mathbf{Add}(\mathsf{ct_{in}}, \mathsf{ct_{index_{best}}})$
return $\mathsf{ct_{out}} = \mathsf{ct_{DriftMitigation}}$

$$\mathsf{Var}(\mathsf{MS}) = \left(\sigma_{\mathsf{in}}^2 + \sigma_{\mathsf{index}}^2\right)\frac{w^2}{q^2} + \frac{1}{24} + c_{var}\frac{w^2}{2q^2} + \frac{n}{2}\left(\left(\frac{1}{12} + c_{var}\frac{w^2}{q^2}\right)\mathsf{Var}(s_i) + \left(\frac{1}{12} + c_{exp}\frac{w^2}{q^2}\right)\mathbb{E}^2(s_i)\right). \tag{15}$$

Here again c_{var} and c_{exp} are the same small constants that depend on the exact implementation of the rounding operation[5]*.*

The complexity of the modulus switch without drift mitigation is:

$$\mathbb{C}^{\mathsf{MS}} = (n+1)\cdot\mathbb{C}^{(q,w)}(\mathsf{Round}). \tag{16}$$

Where $\mathbb{C}^{(q,w)}(\mathsf{Round})$ denotes the complexity of performing a modulus switch from q to w on a single element of $\mathbb{Z}_q$. If the average number of encryptions of zero we need to try before satisfying the condition $|\tilde{\mu}| + \mathsf{r}\cdot\tilde{\sigma} \geq \mathsf{T}$ of the drift mitigation is given by avg[6]*, the complexity of the modulus switch with drift mitigation is on average:*

$$\mathbb{C}^{\mathsf{MS}} = \mathsf{avg}\cdot(n+1)\cdot\mathbb{C}^{(q,w)}(\mathsf{Round}) + \mathsf{avg}\cdot\mathbb{C}(\mathsf{Add}). \tag{17}$$

2.6.2 External Product

In Algorithm 16, we describe the external product. The external product takes as inputs a GGSW ciphertext (Definition 8) encrypting a plaintext $M \in \mathfrak{R}_{q,N}$ and a GLWE ciphertext (Definition 4) encrypting the encoded message $\widetilde{M}$ and returns a GLWE encryption of $M\cdot\widetilde{M}$.

Algorithm 16: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{ExternalProduct}\left(\mathsf{CT}_{\mathsf{in}}, \overline{\overline{\mathsf{CT}}}\right)$

Context:
$\begin{cases} \mathbf{S} = (S_0,\cdots,S_{k-1}) \in \mathfrak{R}_{q,N}^k : \text{the GLWE secret key} \\ \mathsf{CT}_{\mathsf{in}} = (A_0,\cdots,A_{k-1},B) \in \mathfrak{R}_{q,N}^{k+1} : \text{ the input GLWE ciphertext} \\ \overline{\overline{\mathsf{CT}}} = \left(\overline{\mathsf{CT}}_0,\cdots,\overline{\mathsf{CT}}_k\right) \in \mathfrak{R}_{q,N}^{(k+1)\times\ell\times(k+1)} : \text{the input GGSW ciphertext} \\ \ell : \text{ the decomposition level} \\ \mathfrak{B} : \text{ the decomposition base} \end{cases}$

Input:
$\begin{cases} \mathsf{CT}_{\mathsf{in}} \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M}\right), \text{ with } \widetilde{M} \in \mathfrak{R}_{q,N} \\ \overline{\overline{\mathsf{CT}}} \in \mathsf{GGSW}_{\mathbf{S}}^{\mathfrak{B},\ell}(M), \text{ with } M \in \mathfrak{R}_{q,N} \end{cases}$

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathsf{GLWE}_{\mathbf{S}}\left(M\cdot\widetilde{M}\right)$

$\mathsf{CT}_{\mathsf{out}} \leftarrow \left\langle \overline{\mathsf{CT}}_k, \mathsf{BalDec}^{(\mathfrak{B},\ell)}(B)\right\rangle$
for $0 \leq i \leq k-1$ **do**
  $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{CT}_{\mathsf{out}} + \left\langle \overline{\mathsf{CT}}_i, \mathsf{BalDec}^{(\mathfrak{B},\ell)}(A_i)\right\rangle$
return $\mathsf{CT}_{\mathsf{out}}$

Theorem 9 (External Product) *Let $\mathsf{CT}_{\mathsf{in}} \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M}\right) \subseteq \mathfrak{R}_{q,N}^{k+1}$ be a GLWE ciphertext encrypting the encoded message $\widetilde{M} \in \mathfrak{R}_{q,N}$, under the GLWE secret key $\mathbf{S} = (S_0,\ldots,S_{k-1}) \in \mathfrak{R}_{q,N}^{k+1}$, with noise coefficients which can be viewed as having been sampled from χ_σ.*

Let $M \in \mathfrak{R}_{q,N}$ be a message and $\overline{\overline{\mathsf{CT}}} = \left\{\overline{\mathsf{CT}}_i \in \mathsf{GLev}_{\mathbf{S}}^{\mathfrak{B},\ell}(-M\cdot S_i) \subseteq \mathfrak{R}_{q,N}^{\ell\times(k+1)}\right\}_{0\leq i\leq k-1}$ be a GGSW ciphertext with noise sampled from $\chi_{\sigma_{\mathsf{BSK}}}$ and where $S_k = -1$.

[5] An example of a rounding operation and its corresponding concrete values for c_{var} and c_{exp} are given in Remark 19.
[6] This value can be computed based on formula 6.2 of [BJSW24].

In the special case where M is a constant polynomial drawn uniformly from $\{0,1\}$ and $q/\mathfrak{B}^\ell$ is even, the variance of the noise of the coefficients after the external product is:

$$\begin{aligned}\mathsf{Var}(\mathsf{ExternalProduct}) = \frac{\sigma^2}{2} + \ell\cdot(k+1)\cdot N\cdot\left(\frac{\mathfrak{B}^2}{12}+\varepsilon(\mathfrak{B},\ell)\right)\cdot\sigma^2_{\mathsf{BSK}} + \frac{q^2}{24\mathfrak{B}^{2\ell}} + \frac{c_{var}}{2} \\ + kN\cdot\left(\left(\frac{q^2}{24\mathfrak{B}^{2\ell}}+\frac{c_{var}}{2}\right)\mathsf{Var}(s_i) + \left(\frac{q^2}{24\mathfrak{B}^{2\ell}}+\frac{c_{exp}}{2}\right)\mathbb{E}^2(s_i)\right) \\ + \mathsf{FFTNoiseVar}(\mathfrak{B},\ell,k,N)\end{aligned} \tag{18}$$

where $\mathbb{E}(s_i)$ and $\mathsf{Var}(s_i)$ are the expectation and the variance of the secret key coefficients. The term $\mathsf{FFTNoiseVar}(\mathfrak{B},\ell,k,N)$ denotes the noise which is added due to the non-exact nature of the floating point implementation of the FFT algorithm that is used to perform polynomial multiplication; cf. Remark 22.

The complexity of the algorithm is:

$$\begin{aligned}\mathbb{C}^{\mathsf{ExternalProduct}} &= (k+1)N\mathbb{C}^{(\ell)}(\mathsf{BalDec}) + \ell(k+1)\mathbb{C}^{(N)}(\mathsf{FFT}) \\ &+ (k+1)\ell(k+1)N\mathbb{C}^{(N)}(\mathsf{multFFT}) \\ &+ (k+1)(\ell(k+1)-1)N\mathbb{C}^{(N)}(\mathsf{addFFT}) \\ &+ (k+1)\mathbb{C}^{(N)}(\mathsf{iFFT})\end{aligned} \tag{19}$$

Where $\mathbb{C}^{(N)}(\mathsf{FFT}), \mathbb{C}^{(N)}(\mathsf{iFFT})$ denote the cost of performing a size N FFT and inverse FFT, respectively, and $\mathbb{C}^{(N)}(\mathsf{addFFT}), \mathbb{C}^{(N)}(\mathsf{multFFT})$ denotes the cost of performing addition and multiplication of size N in the Fourier domain.

Remark 22 (FFT Noise) *As mentioned in the above theorem, we assumed that the FFT algorithm was used to perform polynomial multiplication, and as such the non-exact nature of the implementation of this algorithm adds a noise term to the output ciphertext. In particular, this means the FFT noise is implementation-dependent and can be the dominant noise term depending on the implementation and the parameters used. Due to this fact, we will not state the formula for the FFT noise variance, $\mathsf{FFTNoiseVar}$, here explicitly. If an exact method for polynomial multiplication is used, then the variance of the error after the external product would not contain the $\mathsf{FFTNoiseVar}(\mathfrak{B},\ell,k,N)$ term.*

In the following Theorems, we will assume again that the FFT is used whenever we need to perform polynomial multiplication. As such, the same comments also apply to these scenarios as well.

2.6.3 CMUX

The CMux is a homomorphic selector. Given two GLWE ciphertexts encrypting two encoded polynomial messages $\widetilde{M_0}$ and $\widetilde{M_1}$ and a GGSW ciphertext encrypting a single bit β (the selector bit), the CMux returns a GLWE encryption of $\widetilde{M_\beta}$. Intuitively, the algorithm homomorphically computes $\widetilde{M_\beta} = (\widetilde{M_1} - \widetilde{M_0})\cdot\beta + \widetilde{M_0}$.

This algorithm is built on top of the external product (Algorithm 16 and Theorem 9) and is the cornerstone of TFHE's PBS. The algorithm is detailed in Algorithm 17 and in the following theorem, we give the noise formula for a CMux.

Theorem 10 (CMUX) *For $i\in\{0,1\}$, let $\mathsf{CT}_i \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M_i}\right) \subseteq \mathfrak{R}_{q,N}^{k+1}$ be two GLWE ciphertexts encrypting encoded messages $\widetilde{M_i}\in\mathfrak{R}_{q,N}$ under the same GLWE secret key $\mathbf{S}=(S_0,\ldots,S_{k-1})\in\mathfrak{R}_{q,N}^{k+1}$, with noise whose coefficients can be seen as having been sampled from χ_σ. Let $\beta\in\{0,1\}$ be a bit. Let $\overline{\overline{\mathsf{CT}}} = \left\{\overline{\mathsf{CT}}_i \in \mathsf{GLev}^{\mathfrak{B},\ell}_{\mathbf{S}}(-\beta\cdot S_i)\subseteq\mathfrak{R}_{q,N}^{\ell\times(k+1)}\right\}_{0\le i\le k}$ be a GGSW with noise sampled from $\chi_{\sigma_{\mathsf{BSK}}}$.*

Algorithm 17: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{CMux}\Big(\mathsf{CT}_0, \mathsf{CT}_1, \overline{\overline{\mathsf{CT}}}\Big)$

Context: $\begin{cases} \mathbf{S} \in \mathfrak{R}_{q,N}^k : \text{the GLWE secret key} \\ \ell : \text{ the decomposition level} \\ \mathfrak{B} : \text{ the decomposition base} \end{cases}$

Input: $\begin{cases} \mathsf{CT}_0 \in \mathsf{GLWE}_{\mathbf{S}}\Big(\widetilde{M_0}\Big), \text{ with } \widetilde{M_0} \in \mathfrak{R}_{q,N} \\ \mathsf{CT}_1 \in \mathsf{GLWE}_{\mathbf{S}}\Big(\widetilde{M_1}\Big), \text{ with } \widetilde{M_1} \in \mathfrak{R}_{q,N} \\ \overline{\overline{\mathsf{CT}}} \in \mathsf{GGSW}_{\mathbf{S}}^{\mathfrak{B},\ell}(\beta), \text{ with } \beta \in \{0,1\} \end{cases}$

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathsf{GLWE}_{\mathbf{S}}\Big(\widetilde{M_\beta}\Big)$

$\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{ExternalProduct}\Big(\mathsf{CT}_1 - \mathsf{CT}_0, \overline{\overline{\mathsf{CT}}}\Big) + \mathsf{CT}_0$

return $\mathsf{CT}_{\mathsf{out}}$

The variance of the noise coefficients of $\mathsf{CT}_{\mathsf{out}}$ *in Algorithm 17, assuming again that* $q/\mathfrak{B}^\ell$ *is even:*

$$\begin{aligned} \mathsf{Var}(\mathsf{CMux}) = \sigma^2 &+ \ell \cdot (k+1) \cdot N \cdot \left(\frac{\mathfrak{B}^2}{12} + \varepsilon(\mathfrak{B}, \ell)\right) \cdot \sigma^2_{\mathsf{BSK}} + \frac{q^2}{24\mathfrak{B}^{2\ell}} + \frac{c_{var}}{2} \\ &+ kN \cdot \left(\left(\frac{q^2}{24\mathfrak{B}^{2\ell}} + \frac{c_{var}}{2}\right)\mathsf{Var}(s_i) + \left(\frac{q^2}{24\mathfrak{B}^{2\ell}} + \frac{c_{exp}}{2}\right)\mathbb{E}^2(s_i)\right) \\ &+ \mathsf{FFTNoiseVar}(\mathfrak{B}, \ell, k, N) \end{aligned} \tag{20}$$

The complexity of Algorithm 17 is:

$$\begin{aligned} \mathbb{C}^{\mathsf{CMux}} = &(k+1)N \cdot \mathbb{C}^{(\ell)}(\mathsf{BalDec}) + \ell(k+1) \cdot \mathbb{C}^{(N)}(\mathsf{FFT}) \\ &+ (k+1)\ell(k+1)N \cdot \mathbb{C}^{(N)}(\mathsf{multFFT}) \\ &+ (k+1)(\ell(k+1)-1)N \cdot \mathbb{C}^{(N)}(\mathsf{addFFT}) \\ &+ (k+1) \cdot \mathbb{C}^{(N)}(\mathsf{iFFT}) + 2 \cdot (k+1)N \cdot \mathbb{C}(\mathsf{add}). \end{aligned} \tag{21}$$

2.6.4 Blind Rotate

The blind rotate algorithm is one of the three main building blocks of the PBS, it is by far the most complex and costly part. It consists in homomorphically rotating a lookup table. Given a GLWE ciphertext CT encrypting an encoded message $\widetilde{M}$ and an LWE ciphertext modulo $2N$ encrypting an encoded message $\widetilde{m}$ with noise e, it returns a GLWE ciphertext encrypting $\widetilde{M} \cdot X^{-\widetilde{m}-e} \in \mathfrak{R}_{q,N}$, i.e., we rotate one ciphertext by the (negative) phase of another. We know that the constant term of $\widetilde{M} \cdot X^{-\widetilde{m}-e}$ is the $(\widetilde{m}+e \bmod N)$-th coefficient of $\widetilde{M}$ up to sign, with the sign being positive if $0 \leq \widetilde{m}+e < N$ and the sign being negative otherwise.

We need to introduce the concept of a redundant lookup table to deal with the noise term e. The motivation behind these lookup tables will be explained in the next part when we will introduce the PBS but essentially they are used to perform the decoding operation.

Definition 13 (Redundant Lookup Table) *A* redundant LUT *is a lookup table encoding a function* f*, whose entries are redundantly represented inside the coefficients of a polynomial in* $\mathfrak{R}_{q,N}$*. In practice, the redundancy consists in an* r*-times (with* r *a system parameter) repetition of the entries* $f(i)$ *of the* LUT *with a certain shift. We use the term* mega-cell *for each block of successive redundant values in an* r*-redundant lookup table:*

Algorithm 18: $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{BlindRotate}(\mathsf{ct}_{\mathsf{in}}, \mathsf{BSK}, \mathsf{CT}_f)$

Context: $\begin{cases} \mathbf{s} = (s_0, \cdots, s_{n-1}) \in \mathbb{Z}^n\text{: the LWE input secret key with } s_i \xleftarrow{\$} \mathcal{U}(\{0,1\}) \\ \mathbf{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N}^k\text{: the output GLWE secret key} \\ \widetilde{M}_f \in \mathfrak{R}_{q,N}\text{: a redundant LUT for a function: } x \mapsto f(x) \end{cases}$

Input: $\begin{cases} \mathsf{ct}_{\mathsf{in}} = (a_0, \cdots, a_{n-1}, b) \in \mathsf{LWE}_{\mathbf{s}}(\widetilde{m}) \subseteq \mathbb{Z}_{2N}^{n+1}, \text{ for some } \widetilde{m} \in \mathbb{Z}_{2N} \\ \mathsf{BSK} = \left\{\overline{\overline{\mathsf{CT}}}_i \in \mathsf{GGSW}_{\mathbf{S}}^{\mathfrak{B},\ell}(s_i)\right\}_{i=0}^{n-1}\text{: a bootstrapping key from } \mathbf{s} \text{ to } \mathbf{S} \\ \mathsf{CT}_f \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M}_f\right) \in \mathfrak{R}_{q,N}^{k+1}\text{: an encrypted (possibly trivially) redundant LUT} \end{cases}$

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M}_f \cdot X^{-\widetilde{m}-e}\right)$, where e is the noise in $\mathsf{ct}_{\mathsf{in}}$

$\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{CT}_f \cdot X^{-b}$
for $0 \leq i \leq n-1$ **do**
  $\mathsf{CT}_0 \leftarrow \mathsf{CT}_{\mathsf{out}}$
  $\mathsf{CT}_1 \leftarrow \mathsf{CT}_{\mathsf{out}} \cdot X^{a_i}$
  $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{CMux}\left(\mathsf{CT}_0, \mathsf{CT}_1, \overline{\overline{\mathsf{CT}}}_i\right)$
return $\mathsf{CT}_{\mathsf{out}}$

$$\widetilde{M}_f = X^{-r/2} \cdot \sum_{i=0}^{N/r-1} X^{i \cdot r} \cdot \left(\sum_{j=0}^{r-1} \widetilde{f(i)} \cdot X^j \right) \tag{22}$$

with $\widetilde{f(i)}$ an encoding of a message $f(i)$. Sometimes we will write this lookup table using just the mega-cell values as

$$\left[\widetilde{f(0)}, \cdots, f\widetilde{\left(\tfrac{N}{r}-1\right)}\right].$$

The redundancy is used to perform the rounding operation needed to remove the noise during bootstrapping.

The blind rotate operation requires some public key material, which we call the bootstrapping key BSK. Informally, the bootstrapping key is a collection of GGSW ciphertexts, one for each element of the input LWE secret key. Formally, we give the following definition:

Definition 14 *Let $\mathfrak{B}$ and ℓ be a pair of decomposition parameters, i.e., a decomposition base and level. Given an input binary LWE secret key $\mathbf{s} = (s_0, \cdots, s_{n-1})$ and an output GLWE secret key $\mathbf{S}$, the corresponding bootstrapping key is the set of GGSW ciphertexts:*

$$\mathsf{BSK} = \left\{\overline{\overline{\mathsf{CT}}} \in \mathsf{GGSW}_{\mathbf{S}}^{\mathfrak{B},\ell}(s_i)\right\}_{i=0}^{n-1}.$$

We describe the blind rotate in Algorithm 18 and in Theorem 11.

Theorem 11 (Blind Rotate) *Let $\mathsf{ct}_{\mathsf{in}} \in \mathsf{LWE}_{\mathbf{s}}(\widetilde{m})$ be an LWE ciphertext encrypting an encoded message $\widetilde{m}$ with ciphertext modulus $2N$ and LWE secret key $\mathbf{s} = (s_0, \cdots, s_{n-1}) \in \mathbb{Z}^n$ with all s_i binary. Let $\mathsf{CT}_f \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M}_f\right) \subseteq \mathfrak{R}_{q,N}^{k+1}$ be a GLWE ciphertext encrypting a redundant LUT for a function f with $\mathbf{S} \in \mathfrak{R}_{q,N}^k$. Let BSK be a bootstrapping key from $\mathbf{s}$ to $\mathbf{S}$ with decomposition parameters $\mathfrak{B}$ and ℓ, i.e., $\mathsf{BSK} = \left\{\overline{\overline{\mathsf{CT}}}_i \in \mathsf{GGSW}_{\mathbf{S}'}^{\mathfrak{B},\ell}(s_i)\right\}_{i=0}^{n-1}$. Suppose further that the noises in all the GLWE ciphertexts in BSK have coefficients independently drawn from a distribution $\chi_{\sigma_{\mathsf{BSK}}}$. Also, suppose that*

the noise in $\mathsf{ct_{in}}$ *is* e *and that the noise in* CT_f *can be seen as having been sampled from a distribution* χ_{σ_f}.

Algorithm 18 outputs a ciphertext $\mathsf{CT_{out}} \in \mathsf{GLWE_S}\left(\widetilde{M}_f \cdot X^{-\widetilde{m}-e}\right)$ *and the variance of the coefficients of the noise in* $\mathsf{CT_{out}}$ *is:*

$$\begin{aligned}\mathsf{Var}(\mathsf{BR}) = \sigma_f^2 &+ n \cdot \left(\ell \cdot (k+1) \cdot N \cdot \left(\frac{\mathfrak{B}^2}{12} + \varepsilon(\mathfrak{B}, \ell)\right) \cdot \sigma^2_{\mathsf{BSK}} + \frac{q^2}{24\mathfrak{B}^{2\ell}} + \frac{c_{var}}{2}\right. \\ &\left. + kN \cdot \left(\left(\frac{q^2}{24\mathfrak{B}^{2\ell}} + \frac{c_{var}}{2}\right)\mathsf{Var}(S_{i,j}) + \left(\frac{q^2}{24\mathfrak{B}^{2\ell}} + \frac{c_{exp}}{2}\right)\mathbb{E}^2(S_{i,j})\right)\right) \\ &+ n \cdot \mathsf{FFTNoiseVar}(\mathfrak{B}, \ell, k, N)\end{aligned} \tag{23}$$

where $\mathsf{Var}(S_{i,j})$ *and* $\mathbb{E}^2(S_{i,j})$ *are the variance and expectation squared of the GLWE secret key coefficients. Note that the output noise does not depend on the noise in* $\mathsf{ct_{in}}$.

The complexity of the blind rotate is:

$$\mathbb{C}^{\mathsf{BR}} = n \cdot \mathbb{C}^{\mathsf{CMux}} \tag{24}$$

In a PBS, CT_f is typically a trivial encryption of $\widetilde{M}_f$ (Definition 6) so that it can be computed publically from knowledge of f, in this case χ_{σ_f} is the constant zero distribution so $\sigma_f^2 = 0$. If one does not want the function f to be public knowledge, then CT_f needs to be given as part of the public key material for all f that are to be used.

Remark 23 (Interest of the Modulus Switch in the PBS) *To make the blind rotate work (Algorithm 18), we need* $2N$ *to be equal to the ciphertext modulus of the input ciphertext, which is impractical for* $q = 2^{64}$. *As the complexity of the blind rotate depends on* N, *we want* N *as small as possible. To this end, we perform a modulus switch before a blind rotate to reduce the ciphertext modulus. This operation adds a lot of noise but allows to choose polynomial sizes much smaller than the size of* q.

2.7 PBS & Its Variants

The Programmable Bootstrapping (PBS) is a fundamental building block of TFHE, enabling both noise reduction and homomorphic function evaluation on encrypted data. As previously discussed in Remark 23, one of the key challenges in implementing a PBS efficiently is handling the ciphertext modulus. The modulus switch step is crucial in reducing the ciphertext modulus before performing a blind rotation, allowing smaller polynomial sizes while keeping the computational complexity manageable.

In this section, we present the classical PBS and explore its variants, which aim to optimize its performance and extend its capabilities. We first formalize the PBS, detailing its major steps: the modulus switch, the blind rotate, and the sample extract. We then introduce several variants, including the multi-bit blind rotate, which enhances parallelizability, and the ManyLUT PBS, which enables multiple function evaluations in a single PBS. Additionally, we discuss alternative approaches to blind rotation and conclude with the concept of atomic patterns, which provide structured ways to compose PBS-based operations efficiently.

2.7.1 Programmable Bootstrap (PBS)

We have introduced in the previous parts all the building blocks needed to explain the PBS. The Programmable Bootstrap operation [CGGI20, CJL+20, CJP21], or PBS, is an FHE algorithm that achieves two goals. First, it resets the noise in a ciphertext to a fixed level (when certain conditions are fulfilled) that is independent of the input noise level. Second, it allows to homomorphically evaluate, at the same time, a lookup table on the encrypted message, i.e., apply an appropriate univariate

function. Such an operator takes as input an LWE ciphertext encrypting an encoding of the message m, a bootstrapping key BSK (Definition 14), an encryption (potentially trivial) of a redundant lookup table $\widetilde{M_f}$ (Definition 13), and outputs an LWE ciphertext with a fixed level of noise encrypting an encoding $\widetilde{f(m)}$ of the message $f(m)$ when the process is successful. When the redundancy matches the encoding used, the algorithm outputs the correct output up to the failure probability p_{fail} that can be estimated using the variance of the noise of the input LWE ciphertext and the parameters used during blind rotate.

The PBS is composed of three major steps: the modulus switch (Algorithm 14 and Theorem 8), the blind rotate (Algorithm 18 and Theorem 11), and the sample extract (Algorithm 7 and Theorem 5).

Algorithm 19: $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{PBS}(\mathsf{CT}_f, \mathsf{ct}_{\mathsf{in}}, \mathsf{BSK}, \mathsf{DriftMitigation}, \mathsf{DMK})$

Context:
- $\mathbf{s} = (s_0, \cdots, s_{n-1}) \in \mathbb{Z}^n$: the input binary LWE secret key
- $\mathbf{S}' = \left(S'_0, \ldots, S'_{k-1}\right) \in \mathfrak{R}_{q,N}^k$: a GLWE secret key where $S'_i = \sum_{j=0}^{N-1} s'_{i\cdot N+j} X^j$ for all i
- $\mathbf{s}' = (s'_0, \cdots, s'_{kN-1}) \in \mathbb{Z}_q^{kN}$: the output LWE secret key
- $\widetilde{M_f} \in \mathfrak{R}_{q,N}$: a redundant LUT for a function f
- $\mathfrak{B}$ and ℓ are decomposition parameters

Input:
- $\mathsf{CT}_f \in \mathsf{GLWE}_{\mathbf{S}'}\left(\widetilde{M_f}\right) \subseteq \mathfrak{R}_{q,N}^{k+1}$
- $\mathsf{ct}_{\mathsf{in}} \in \mathsf{LWE}_{\mathbf{s}}(\widetilde{m}) \subseteq \mathbb{Z}_q^{n+1}$, where $\widetilde{m} = \Delta \cdot m$
- BSK : a bootstrapping key from $\mathbf{s}$ to $\mathbf{S}'$
- $\mathsf{DriftMitigation}$: Boolean value indicating if the drift mitigation should be applied
- DMK : drift mitigation key, only used when $\mathsf{DriftMitigation}$ is True

Output: $\mathsf{ct}_{\mathsf{out}} \in \mathsf{LWE}_{\mathbf{s}'}\left(\widetilde{f(m)}\right)$ if we respect the requirements of Theorem 12

1 $\mathsf{ct}_{\mathsf{MS}} \leftarrow \textbf{ModulusSwitch}(\mathsf{ct}_{\mathsf{in}}, 2N, \mathsf{DriftMitigation}, \mathsf{DMK})$
2 $\mathsf{CT} \leftarrow \textbf{BlindRotate}(\mathsf{ct}_{\mathsf{MS}}, \mathsf{BSK}, \mathsf{CT}_f)$
3 $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{SE}(\mathsf{CT}, 0)$
4 **return** $\mathsf{ct}_{\mathsf{out}}$

Theorem 12 (Programmable Bootstrap (PBS)) *Let $\mathbf{s} = (s_0, \cdots, s_{n-1}) \in \mathbb{Z}^n$ be a binary LWE secret key. Let $\mathbf{S}' = \left(S'_0, \ldots, S'_{k-1}\right) \in \mathfrak{R}_{q,N}^k$ be a GLWE secret key such that $S'_i = \sum_{j=0}^{N-1} s'_{i\cdot N+j} \cdot X^j$, and denote by $\mathbf{s}' = (s'_0, \cdots, s'_{kN-1})$ the corresponding flattened LWE secret key. Let BSK be the associated bootstrapping key (Definition 14) from $\mathbf{s}$ to $\mathbf{S}'$. Assume further that q is a power of two larger than N.*

Suppose we have an LWE ciphertext $\mathsf{ct}_{\mathsf{in}} \in \mathsf{LWE}_{\mathbf{s}}(\widetilde{m})$ where $\widetilde{m}$ is the encoding $\widetilde{m} = \Delta \cdot m$ of a message $m \in \mathbb{Z}_{2p}$ for some p dividing N and $\Delta = \frac{q}{2p}$, and suppose the noise in $\mathsf{ct}_{\mathsf{in}}$ can be viewed as having been sampled from the noise distribution $\chi_{\sigma_{\mathsf{in}}}$.

Let $\widetilde{M_f}$ be the $\frac{N}{p}$-redundant lookup-table (Definition 13) for a function $f\colon \mathbb{Z}_p \to \mathbb{Z}_q$ and let CT_f be a (potentially trivial) GLWE encryption of $\widetilde{M_f}$.

Then Algorithm 19 outputs an LWE ciphertext $\mathsf{ct}_{\mathsf{out}}$ under the secret key $\mathbf{s}'$ encrypting $\widetilde{f(m)}$, i.e., an encoding of $f(m)$ with a probability $1 - p_{\mathsf{fail}}$ if and only if both

- *the input noise has variance satisfying*

$$\sigma_{\mathsf{in}}^2 < \frac{\Delta^2}{4 \cdot z^*(p_{\mathsf{fail}})^2} - \frac{1}{1+D}\left(\frac{q^2}{48N^2} + c_{var} + n\left(\frac{q^2}{96N^2} + \frac{c_{var} + c_{exp}}{4}\right)\right) - D \cdot \sigma_{\mathsf{index}}^2,$$

  where $D = 1$ if $\mathsf{DriftMitigation}$ is true, else $D = 0$, and where $z^*(p_{\mathsf{fail}})$ is the standard score associated to the failure probability p_{fail} given by

$$z^*(p_{\mathsf{fail}}) = \sqrt{2} \cdot \mathrm{erf}^{-1}(1 - p_{\mathsf{fail}})$$

and erf^{-1} *is the inverse of the error function* $\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} \mathrm{d}t$*;*

- *and either* $0 \leq m < p$ *or both* $p \leq m < 2p$ *and* $f(m) = -f(m-p)$ *holds.*

The output noise after the PBS *is that of the blind rotate operation, that is we have* $\mathsf{Var}(\mathsf{PBS}) = \mathsf{Var}(\mathsf{BlindRotate})$ *from Eq.* (23)*.*

The complexity of Algorithm 19 is:

$$\mathbb{C}^{\mathsf{PBS}} = \mathbb{C}^{\mathsf{MS}} + \mathbb{C}^{\mathsf{BR}} + \mathbb{C}^{\mathsf{SE}}. \tag{25}$$

The size (in bits) of the traditional bootstrapping key $\mathsf{Size}(\mathsf{BSK})$ *is computed with the following formula:*

$$\mathsf{Size}(\mathsf{BSK}) \coloneqq n \cdot \ell_{\mathsf{PBS}} \cdot (k+1)^2 \cdot N \cdot \lceil \log_2(q) \rceil.$$

Remark 24 (Negacyclic Function) *The second condition in Theorem 12 essentially states that we can only use half the message space* ($\mathbb{Z}_{2p}$) *when applying a* PBS *with the exception that we can use the full message space if* $f \colon \mathbb{Z}_{2p} \to \mathbb{Z}_q$ *has the following property:*

$$f(x+p) = -f(x) \text{ for all } x \in \mathbb{Z}_{2p}. \tag{26}$$

This is why, in general, we need to use at least one bit of padding in our encoding. If f *satisfies Eq.* (26)*, then the function* f *is said to be negacyclic and we do not require a padding bit for the PBS to work. For odd message modulus (see Definition 9), it is also possible to use an arbitrary* f *without using a padding bit.*

Remark 25 (Independence of input and output encoding) *We point out that the encoding used in the input LWE of the PBS and the encoding used in the output LWE (determined via the encoding in the redundant* LUT*) are independent from one another. They can differ in their respective number of padding bits* π *and in their respective message modulus* p*. The output encoding can even use a completely different style of encoding if so desired.*

Remark 26 (General ciphertext modulus) *Theorem 12 assumes that the ciphertext modulus* q *is a power of two, this restriction can be lifted without any change to Algorithm 19 but the noise from both the modulus switch and the blind rotate will be slightly different.*

Furthermore, we do not actually require that the ciphertext modulus of the input LWE is q *as long as we still correctly modulus switch this ciphertext to modulus* $2N$ *in the first step. For example, one can use a PBS to change the ciphertext modulus from say* 2^{64} *to* 2^{128} *using a bootstrapping key with modulus* 2^{128}*.*

Remark 27 (Key Switch and PBS) *More often than not, we apply a key switch before a* PBS *with the idea that we keyswitch to a smaller LWE dimension and thus have to perform fewer* CMux *operations during the blind rotate. For simplicity, we will sometimes refer to the sequential combination of a key switch followed by a* PBS *as a* KS-PBS*.*

Remark 28 (Using a carry space to enable multivariate function evaluation) *Natively, the PBS operation can only apply a univariate function, however by using an encoding with a carry space it is possible to homomorphically compute a bivariate (or even multivariate) function such as multiplication. This trick was proposed in [CZB+22] and works for a suitably large carry space. Assuming the carry spaces have been emptied, the idea is to concatenate two messages* m_1 *and* m_2 *(or more) respectively encrypted in* ct_1 *and* ct_2 *by rescaling the first one with constant multiplication by* μ_2+1 *(where* μ_2 *is the largest possible value that can be taken by* m_2*) and adding it to* ct_2 *and finally computing a PBS on the result. Once the two messages are concatenated into a single ciphertext, the bivariate function*

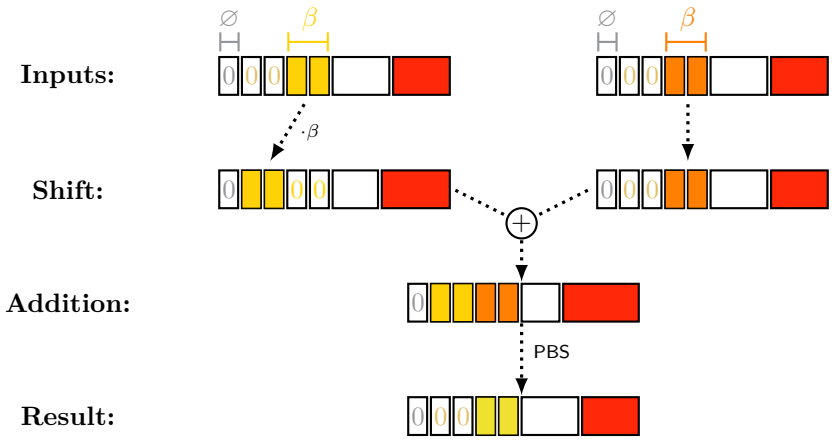

Figure 6: Example of a bivariate function evaluation via a shift, an addition, and a PBS.

can be simply evaluated as a univariate function on the concatenation of m_1 and m_2. A visual example is given in Fig. 6. As mentioned, this gives one way to be able to perform a multiplication using a single PBS operation. Regarding the functions computed in a bivariate bootstrapping, any bivariate function g can be written as a univariate one: $g(a,b) = f(a + b \cdot m)$, with f defined as in Remark 24, $a \in \mathbb{N}_{\mathsf{msg}}$ and $b \in \mathbb{N}_{\mathsf{carry}}$. This gives the possibility of easily reusing the previous encoding for LUT*. The signature of the algorithm is:*

$$\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{BivPBS}(\mathsf{CT}_f, \mathsf{ct}_{\mathsf{in}_1}, \mathsf{ct}_{\mathsf{in}_0}, \mathsf{BSK}, \mathsf{DriftMitigation}, \mathsf{DMK})$$

2.7.2 Multi-Bit Blind Rotate

We note that the blind rotation step of the PBS is not well suited to benefit from parallelism. In this section, we give an alternative blind rotation algorithm that can make the PBS operation more easily parallelizable. We refer to this variant as the *Multi-Bit Blind Rotate* and it was introduced in [ZYL+18]. It performs the blind rotation in chunks, whose length is referred to as the *grouping factor* and denoted gf. This variant is practically employed in the GPU implementation of the Blind Rotate algorithm thanks to its better parallelizability, although its overall cost is higher than that of the vanilla variant described in the previous section, which is used in the CPU implementation. Furthermore, the size of the multi-bit bootstrapping key grows exponentially with the grouping factor.

Likewise to the vanilla variant, the multi-bit one aims to blindly multiply the input GLWE, which holds the LUT, by $X^{\sum a_i s_i}$. However, instead of applying one key bit per step via CMux (cf. Line 5 of Algorithm 18), here we apply a chunk of gf bits via another update step (cf. Line 3 of Algorithm 20). For instance, for $\mathsf{gf} = 2$, the expanded multi-bit update step writes

$$\mathsf{CT}_{\mathsf{out}} \leftarrow \left(X^0 \cdot \overline{\overline{\mathsf{CT}}}_{i,\emptyset} + X^{a_{2i+0}} \cdot \overline{\overline{\mathsf{CT}}}_{i,\{0\}} + X^{a_{2i+1}} \cdot \overline{\overline{\mathsf{CT}}}_{i,\{1\}} + X^{a_{2i+0}+a_{2i+1}} \cdot \overline{\overline{\mathsf{CT}}}_{i,\{0,1\}}\right) \boxdot \mathsf{CT}_{\mathsf{out}} \quad (27)$$

where $\overline{\overline{\mathsf{CT}}}_{i,\emptyset} \in \mathsf{GGSW}_{\mathbf{S}}^{\mathfrak{B},\ell}\big((1-s_{2i+0})\cdot(1-s_{2i+1})\big)$, $\overline{\overline{\mathsf{CT}}}_{i,\{0\}} \in \mathsf{GGSW}_{\mathbf{S}}^{\mathfrak{B},\ell}\big(s_{2i+0}\cdot(1-s_{2i+1})\big)$, etc.

In general, for a fixed i, note that exactly one of $\left\{\overline{\overline{\mathsf{CT}}}_{i,J}\right\}_{J\subseteq\{0,\ldots,\mathsf{gf}-1\}}$ encrypts 1: this is for $\bar{J}$ such that $\prod_{j\in\bar{J}} s_{i\cdot\mathsf{gf}+j} \prod_{j\notin\bar{J}}(1-s_{i\cdot\mathsf{gf}+j}) = 1$, i.e., $\bar{J} = \{j \mid s_{i\cdot\mathsf{gf}+j} = 1\}$; the others encrypt zero. It follows that the new $\mathsf{CT}_{\mathsf{out}}$ encrypts the plaintext of the old $\mathsf{CT}_{\mathsf{out}}$ multiplied by $X^{\sum_{j\in\bar{J}} a_{i\cdot\mathsf{gf}+j}} = X^{\sum_{j=0}^{\mathsf{gf}-1} a_{i\cdot\mathsf{gf}+j} s_{i\cdot\mathsf{gf}+j}}$.

Algorithm 20: $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{MultiBitBlindRotate}(\mathsf{ct}_{\mathsf{in}}, \mathsf{MB\text{-}BSK}, \mathsf{CT}_f)$

Context:
$\begin{cases} \mathsf{gf} \in \mathbb{N}\text{: grouping factor, typically a small integer} \in \{2,3,4\}, \text{ where } n = \mathsf{gf} \cdot y \text{ for some } y \in \mathbb{N} \\ \mathbf{s} = (s_0, \cdots, s_{n-1}) \in \mathbb{Z}^n\text{: the LWE input secret key with } s_i \xleftarrow{\$} \mathcal{U}(\{0,1\}) \\ \mathbf{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N}^k\text{: the output GLWE secret key} \\ \widetilde{M}_f \in \mathfrak{R}_{q,N}\text{: a redundant } \mathsf{LUT} \text{ for a function } x \mapsto f(x) \end{cases}$

Input:
$\begin{cases} \mathsf{ct}_{\mathsf{in}} = (a_0, \cdots, a_{n-1}, b) \in \mathsf{LWE}_{\mathbf{s}}(\widetilde{m}) \subseteq \mathbb{Z}_{2N}^{n+1}, \text{ for some } \widetilde{m} \in \mathbb{Z}_{2N} \\ \mathsf{MB\text{-}BSK} = \left\{ \left\{ \overline{\overline{\mathsf{CT}}}_{i,J} \in \mathsf{GGSW}_{\mathbf{S}}^{\mathfrak{B},\ell}\left(\prod_{j \in J} s_{i \cdot \mathsf{gf}+j} \prod_{j \notin J} (1 - s_{i \cdot \mathsf{gf}+j}) \right) \right\}_{J \subseteq \{0,\ldots,\mathsf{gf}-1\}} \right\}_{i=0}^{y-1} : \\ \qquad \text{a multi-bit bootstrapping key from } \mathbf{s} \text{ to } \mathbf{S} \\ \mathsf{CT}_f \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M}_f\right) \subseteq \mathfrak{R}_{q,N}^{k+1} \text{ : an encrypted (possibly trivially) redundant } \mathsf{LUT} \end{cases}$

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathsf{GLWE}_{\mathbf{S}}\left(\widetilde{M}_f \cdot X^{-\widetilde{m}-e}\right)$, where e is the noise in $\mathsf{ct}_{\mathsf{in}}$

$\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{CT}_f \cdot X^{-b}$
for $0 \le i \le y-1$ **do**
  $\mathsf{CT}_{\mathsf{out}} \leftarrow \left(\sum_{J \subseteq \{0,\ldots,\mathsf{gf}-1\}} X^{\sum_{j \in J} a_{i \cdot \mathsf{gf}+j}} \cdot \overline{\overline{\mathsf{CT}}}_{i,J} \right) \boxdot \mathsf{CT}_{\mathsf{out}}$
return $\mathsf{CT}_{\mathsf{out}}$

Remark 29 (Smaller Multi-Bit Bootstrapping Key) *Since we know that for a fixed i exactly one of $\overline{\overline{\mathsf{CT}}}_{i,J}$ encrypts 1 and the rest encrypts zero, we know that their sum encrypts one. This can be used to remove the need to provide $\overline{\overline{\mathsf{CT}}}_{i,J}$ for one chosen J, say $J = \emptyset$, and hence reduce the multi-bit bootstrapping key size by $\frac{n}{\mathsf{gf}}$ GGSW ciphertexts in total. See [BMMP18] for more information. For the case of $\mathsf{gf} = 2$, Eq. (27) is replaced by*

$$\mathsf{CT}_{\mathsf{out}} \leftarrow \left((X^{a_{2i}} - 1) \cdot \overline{\overline{\mathsf{CT}}}_{i,\{0\}} + (X^{a_{2i+1}} - 1) \cdot \overline{\overline{\mathsf{CT}}}_{i,\{1\}} + \left(X^{a_{2i}+a_{2i+1}} - 1 \right) \cdot \overline{\overline{\mathsf{CT}}}_{i,\{0,1\}} \right) \boxdot \mathsf{CT}_{\mathsf{out}} + \mathsf{CT}_{\mathsf{out}}.$$

2.7.3 ManyLUT PBS

In Remark 24 we noted that for the PBS to be correct for a function that is not negacyclic we needed at least one bit of padding in our encoding. That is $\pi \ge 1$ in the notation of Definition 9. When we have more than one bit of padding we can compute a PBS with more than one function at once using a single blind rotation. This idea was first proposed in [CLOT21] and we present a slightly modified version of it here.

Suppose we have $\pi > 1$ and functions $f_1, \cdots, f_{2^{\pi-1}}$ which we want to apply to the same encoded message $\widetilde{m} = \Delta \cdot m$ where $m \in \mathbb{Z}_p$, then we can construct a redundant lookup table from the f_i and a suitably large N for which we can apply the blind rotation algorithm a single time and from the resulting GLWE ciphertext we can sample extract $2^{\pi-1}$ LWE ciphertexts which with high probability are encryptions of $\widetilde{f_i(m)}$.

The idea is to define a function $f : \mathbb{Z}_{2^{\pi-1}p} \to \mathbb{Z}_q$ from $f_1, \cdots, f_{2^{\pi-1}}$ as follows:

$$f(x) = f_{y+1}(z), \text{ where } x = p \cdot y + z \text{ and } 0 \le y < 2^{\pi-1},\ 0 \le z < p, \tag{28}$$

and note for the lookup table associated to f that every p mega-cells we go from $f_i(z)$ to $f_{i+1}(z)$ for some $0 \le z < p$. During blind rotation we will rotate the lookup table by m mega-cells so that (for small enough noise) sample extracting (with $\alpha = 0$) will give an encryption of $f_1(m)$, sample extracting position $\alpha = N/2^{\pi-1}$ will give an encryption of $f_2(m)$, and so on for all positions that are multiples of $N/2^{\pi-1}$. Details are provided in Algorithm 21.

Algorithm 21: $\left(\mathsf{ct}_{\mathsf{out},1}, \cdots, \mathsf{ct}_{\mathsf{out},2^{\pi-1}}\right) \leftarrow \mathsf{PBSmanyLUT}(\mathsf{ct}_{\mathsf{in}}, \mathsf{BSK}, \mathsf{CT}_f, \mathsf{DriftMitigation})$

Context:
- $\mathbf{s} = (s_0, \cdots, s_{n-1}) \in \mathbb{Z}^n$: the input binary LWE secret key
- $\mathbf{S}' = \left(S'_0, \ldots, S'_{k-1}\right) \in \mathfrak{R}_{q,N}^k$: a GLWE secret key where $S'_i = \sum_{j=0}^{N-1} s'_{i\cdot N+j} X^j$ for all i
- $\mathbf{s}' = (s'_0, \cdots, s'_{kN-1}) \in \mathbb{Z}_q^{kN}$: the output LWE secret key
- $\widetilde{M}_f \in \mathfrak{R}_{q,N}$: a redundant LUT for a function f derived from $f_1, \cdots, f_{2^{\pi-1}}$ using Eq. (28)
- $\mathfrak{B}$ and ℓ are decomposition parameters
- DMK the drift mitigation key

Input:
- $\mathsf{ct}_{\mathsf{in}} \in \mathsf{LWE}_{\mathbf{s}}(\widetilde{m}) \subseteq \mathbb{Z}_q^{n+1}$, where $\widetilde{m} = \Delta \cdot m$ for $m \in \mathbb{Z}_p$ and $\Delta = \lfloor q/(2^\pi \cdot p) \rfloor$
- BSK : a bootstrapping key from $\mathbf{s}$ to $\mathbf{S}'$
- $\mathsf{CT}_f \in \mathsf{GLWE}_{\mathbf{S}'}\left(\widetilde{M}_f\right) \subseteq \mathfrak{R}_{q,N}^{k+1}$
- $\mathsf{DriftMitigation}$: Boolean value indicating if the drift mitigation should be applied

Output: $\mathsf{ct}_{\mathsf{out},i} \in \mathsf{LWE}_{\mathbf{s}'}\left(\widetilde{f_i(m)}\right)$ for $1 \leq i \leq 2^{\pi-1}$ if we respect the requirements of Theorem 12

$\mathsf{ct}_{\mathsf{MS}} \leftarrow \textbf{ModulusSwitch}(\mathsf{ct}_{\mathsf{in}}, 2N, \mathsf{DriftMitigation}, \mathsf{DMK})$
$\mathsf{CT} \leftarrow \textbf{BlindRotate}\left(\mathsf{ct}_{\mathsf{MS}}, \mathsf{BSK}, \mathsf{CT}_f\right)$
for $0 \leq i < 2^\pi$ **do**
  $\mathsf{ct}_{\mathsf{out},i+1} \leftarrow \mathsf{SE}\left(\mathsf{CT}, i \cdot \frac{N}{2^{\pi-1}}\right)$
return $\left(\mathsf{ct}_{\mathsf{out},1}, \cdots, \mathsf{ct}_{\mathsf{out},2^{\pi-1}}\right)$

We remark that this technique works equally well if instead of assuming we have more padding bits we assume we have an empty carry space when using the message-and-carry type encoding. The number of functions one can homomorphically compute simultaneously is the size of the carry space (assuming we also have one bit of padding).

2.7.4 Alternative Approaches to the Blind Rotate

The blind rotate (Section 2.6.4) is the most computationally expensive part of PBS and has been the focus of extensive research for optimization. One such improvement is the multi-bit version presented in Section 2.7.2. The approach in Algorithm 18 only works if the LWE secret key is binary but can be extended to ternary of even larger keys as detailed in [JP22].

In all the approaches we have seen so far, the idea is to start from (a trivial encryption of) X^{a_i} and use the bootstrapping key to compute an encryption of $X^{a_i s_i}$ and multiply the accumulator by the result, this is referred to as the GINX approach [GINX16]. One can instead consider the opposite, starting from an encryption of X^{s_i} (which would be part of the bootstrapping key), compute an encryption of $X^{a_i s_i}$ and multiply the accumulator by it, this approach is the AP style [AP14]. This style naturally benefits from the use of automorphisms and a number of works have explored this [LMK+23, Lee24]. Note that here one must also modify the modulus switch to round to the nearest odd integer as only odd powers can be computed using an automorphism; this idea is pushed even further in [WWL+24].

Another idea to speed up the blind rotate operator is using a non-standard secret key distribution [LMSS23].

2.7.5 Atomic Patterns

As introduced in [BBB+23], the notion of an *atomic pattern* refers to a subgraph of FHE operations that outputs one (or more) ciphertexts. These atomic patterns are typically constructed from basic operations such as the *homomorphic dot product* as defined in Theorem 4, the *key switch* as defined

in Theorem 6, and the *programmable bootstrap* as defined in Theorem 12. The atomic pattern typically used in TFHE-rs is from [CJP21], denoted by $\mathcal{A}^{(\mathsf{CJP})}$, which is composed of a dot product of several input ciphertexts, followed by a keyswitch, and completed with a PBS. An atomic pattern has a matching input and output type, meaning that they can be chained together to build larger computations.

3 Arithmetic over Large Homomorphic Integers

To encode larger integers—i.e., those that do not fit into a single LWE ciphertext (see Section 2.3)—the radix decomposition is used. Programmatically, a radix ciphertext, denoted $\overrightarrow{\mathsf{ct}}$ and referred to as a *homomorphic integer*, is a list of LWE ciphertexts (aka. *blocks*), each encrypting a digit of the decomposition of the original message. Essentially, we implement a homomorphic analogy to the standard multiple precision arithmetic (like, e.g., the GMP Library[7] does in the clear), while instead of cleartext machine words, which typically hold 32 or 64 bits of data, we deal with LWE-encrypted chunks, which only hold small units of bits of data (2 bits in case of the default parameters of TFHE-rs).

In this section, we first describe the encoding in more detail, then we dive into algorithms for individual arithmetic operations.

3.1 Radix Encoding of Large Homomorphic Integers

The radix encoding consists in decomposing a large integer m modulo $\Omega = \prod_{i=0}^{\mathcal{B}-1} \beta_i$ into a list of digits $(m_i)_{i=0}^{\mathcal{B}-1}$, such that $m = m_0 + m_1 \cdot \beta_0 + m_2 \cdot \beta_0 \cdot \beta_1 + \ldots + m_{\mathcal{B}-1} \cdot \frac{\Omega}{\beta_{\mathcal{B}-1}}$ and $0 \leq m_i < \beta_i$. Each digit m_i is individually encoded in the carry-message format introduced in Section 2.3 and encrypted into an LWE ciphertext. Together, these ciphertexts form the homomorphic integer $\overrightarrow{\mathsf{ct}}$, which encrypts m modulo Ω. The encoding of m_i is defined according to a pair $(\beta_i, p_i) \in \mathbb{N}^2$ of parameters, such that $2 \leq \beta_i \leq p_i$, which respectively corresponds to the size of the message subspace and the carry-message space (i.e., in the carry-message paradigm, we have $\beta_i = \mathsf{msg}_i$ and $p_i = \mathsf{msg}_i \cdot \mathsf{carry}_i$); Fig. 7 gives a visual representation of a toy example.

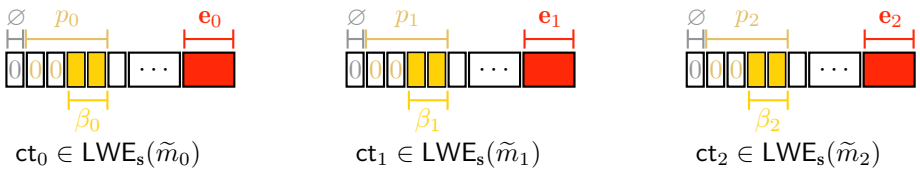

Figure 7: Plaintext representation of a fresh radix-based large homomorphic integer m of length $\mathcal{B} = 3$, where for all $0 \leq i < \mathcal{B}$ we have $\pi = 1$, $\beta_i = 2^2$ and $p_i = 2^{2+2}$, hence working modulo $\Omega = (2^2)^3$. For the encoding, it holds that $m = m_0 + m_1 \cdot 4 + m_2 \cdot 4^2$. The symbol ∅ represents the padding bit needed for the PBS. For each block we have $\widetilde{m_i} = \mathsf{Encode}(m_i, 2^{1+2+2}, q)$.

Definition 15 (Radix-Encoded Large Homomorphic Integer) *Let $\Omega = \prod_{i=0}^{\mathcal{B}-1} \beta_i$, with $\beta_i \in \mathbb{N}_{>1}$ and $\mathcal{B} \in \mathbb{N}$, be a message modulus, $m \in \mathbb{Z}_\Omega$ a message and let $(m_i)_{i=0}^{\mathcal{B}-1}$ be its respective radix decomposition, i.e.,*

$$m = \sum_{i=0}^{\mathcal{B}-1} m_i \cdot \left(\prod_{j=0}^{i-1} \beta_j \right), \tag{29}$$

such that $0 \leq m_i < \beta_i$. Next, let $(p_i)_{i=0}^{\mathcal{B}-1}$ be the carry-message moduli as per Section 2.3. We set $\widetilde{m_i} = \mathsf{Encode}(m_i, 2^\pi \cdot p_i, q)$ with π the number of bits of padding and q the LWE ciphertext modulus. Finally, we encrypt each $\widetilde{m_i}$, obtaining LWE ciphertexts that form a vector $\overrightarrow{\mathsf{ct}}$, referred to as a homomorphic integer.

To get back the original message m, we simply recompose it using Eq. (29) where the m_i values are obtained by decryption and decoding.

[7] The GNU Multiple Precision Arithmetic Library, https://gmplib.org.

Remark 30 *In practice, the restriction for Ω is that it has to be a product of small bases. Indeed, TFHE-like schemes do not scale well when one increases the precision, so the best practice is to keep $p_i \leq 2^8$. Specifically, TFHE-rs employs by default the same parameters for all blocks: $\beta_i = 4$ and $p_i = 16$, i.e., $\mathsf{msg} = \mathsf{carry} = 2^2$, with $\pi = 1$.*

To perform operations on radix-based integers, messages must be encoded and encrypted with the same parameters to ensure compatibility. The majority of the arithmetic operations can be computed by using a schoolbook approach (homomorphically mixing linear operations and PBSes) and by keeping the degree of fullness (Definition 10) smaller than p_i in each block. When a carry subspace is full, it needs to be propagated to the next block: this is done by extracting the carry and the message (naïvely requiring two PBSes) and adding the carry to the next block.

Remark 31 *Signed integers are made using the two's complement representation. Instead of constructing e.g. a 32-bit unsigned integer which is contained in the range $[0, 2^{32} - 1]$, a 32-bit signed integer sits in the range $[-2^{31}, 2^{31} - 1]$. An integer r in this range is constructed as:*

$$r = -r_{31} \cdot 2^{31} + \sum_{i=0}^{30} r_i \cdot 2^i.$$

Remark 32 *The approach of splitting a message into multiple ciphertexts has already been proposed for binary radix decomposition in FHEW [DM15] and TFHE [CGGI20], and for other representations in [BST20], [GBA21], [KÖ22], [CZB+22] and [LMP21]. However, none of them leverage carry buffers to improve efficiency in computations on homomorphic integers—i.e., integers split across multiple ciphertexts—by minimizing the number of bootstrappings.*

In [GBA21], the authors propose two approaches to evaluate a generic LUT *over these multi-ciphertexts inputs, namely the* tree-based *and* chaining-based *approaches (that we shorten as Tree-*PBS *and Chained-*PBS*), however, these methods become more and more inefficient as the number of blocks goes increasing. The Chained-*PBS *method is further generalized in [CZB+22] to any function in exchange for a larger plaintext space.*

Remark 33 *Alternatives to radix-based integer decomposition exist; in particular, decomposition based on the* Chinese Remainder Theorem *(CRT) is possible. However, for simplicity, we omit further details in this paper.*

3.2 Notations and Assumptions

Throughout Section 3, we outline several algorithms on homomorphic integers and consider their complexity. In particular, the complexity values given in this section denote a count of the number of required PBSes to perform the algorithm. Moreover, we assume that the complexity of a PBS is equivalent to that of a bivariate PBS, treating them as identical throughout using the notation $\mathbb{C}^{\mathsf{PBS}}$.

3.2.1 Example: Addition

Let us start with the algorithm computing the addition between two Radix-based ciphertexts (Algorithm 22).

According to the algorithm signature $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathsf{Add}\left(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\right)$, Add takes as input two ciphertext, $\overrightarrow{\mathsf{ct}}^{(1)}$ and $\overrightarrow{\mathsf{ct}}^{(2)}$, and outputs one ciphertext $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$. The algorithm is straightforward: for each of the $\mathcal{B}$ blocks of each homomorphic integer, an addition of LWE ciphertexts is computed, leveraging the inherent linearity of the radix encoding used. The output ciphertext then consists of the resulting LWE ciphertext blocks.

Algorithm 22: $\overrightarrow{\mathsf{ct}}^{(\mathsf{Add})} \leftarrow \mathsf{Add}\left(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\right)$

Input: $\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}$
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{Add})}$
for $i \in [0, \mathcal{B}-1]$ **do**
  $\mathsf{ct}_i^{(\mathsf{Add})} \leftarrow \mathsf{ct}_i^{(1)} + \mathsf{ct}_i^{(2)}$
return $\overrightarrow{\mathsf{ct}}^{(\mathsf{Add})}$

However, many details are hidden in this algorithm, including all cryptographic parameters such as the ciphertext modulus, scaling factor Δ, and others. To ensure algorithmic correctness, these parameters must be the same for all the blocks. However, computational parameters, such as the noise level and the degree of fullness, must also be constrained. For example, after performing an addition, the noise level per block and the degree of fullness deviate from their nominal values, imposing restrictions on subsequent operations. Consider the case of chaining multiple additions, where both the noise level and the degree of fullness eventually reach their maximum values. To ensure computational correctness, we must empty the carry buffers of the LWE ciphertexts and propagate them to the next block—a process called carry propagation—before performing further computations. The cost of this operation is significant and in the case of addition, it even dominates the overall computation time. This leads to a second challenge: how can we benchmark operations when execution times vary from microseconds to milliseconds, depending on whether carry propagation is applied? To maintain consistency, we chose to enforce that the output encoding matches the input encoding in most algorithms described in this section. Carry propagation can be seen as the integer equivalent of Boolean gate bootstrapping in TFHE, resetting the encoding after each operation to preserve composability and enable more realistic benchmarks.

In the following section, we outline the assumptions we make regarding the inputs and outputs of each algorithm described below.

3.2.2 Input/Output Assumptions

To simplify the notations of the large integer arithmetic algorithms in this section, we establish here once and for all the assumptions we make about any large homomorphic integer. As detailed in the previous section, a large homomorphic integer will be represented through a list of LWE ciphertexts, where each one encrypts a digit of the radix decomposition of the large integer. Each LWE ciphertext of the list will be referred to as a block and the number of blocks will be indicated with $\mathcal{B}$. The blocks of the list are stored in little-endian order, that is, the least significant block (i.e., the block that encrypts the least significant bits, called LSBblock) is stored as the first element and the most significant block is stored as the last element of the list (called MSBblock).

As mentioned in Remark 30, all blocks are assumed to use the same cryptographic parameters. An input ciphertext $\overrightarrow{\mathsf{ct}}^{(1)}$ is then denoted as:

$$\overrightarrow{\mathsf{ct}}^{(1)} = \left(\mathsf{ct}_0^{(1)}, \ldots, \mathsf{ct}_{\mathcal{B}-1}^{(1)}\right) \in \bigtimes_{i \in [0, \mathcal{B}-1]} \mathsf{LWE}_{\mathbf{s}^{(1)}}\left(\mathsf{Encode}\left(m_i^{(1)}, 2^{\pi^{(1)}} \cdot \mathsf{msg}^{(1)} \cdot \mathsf{carry}^{(1)}, q^{(1)}\right)\right) \subseteq \mathbb{Z}_{q^{(1)}}^{(n^{(1)}+1) \times \mathcal{B}^{(1)}},$$

and likewise for $\overrightarrow{\mathsf{ct}}^{(2)}$, etc.

Here is the list of assumptions:

- If an algorithm takes as input two (or more) ciphertexts $\overrightarrow{\mathsf{ct}}^{(1)}$ and $\overrightarrow{\mathsf{ct}}^{(2)}$, then:

$$\begin{aligned}\mathsf{msg} &= \mathsf{msg}^{(1)} = \mathsf{msg}^{(2)}\\ \mathsf{carry} &= \mathsf{carry}^{(1)} = \mathsf{carry}^{(2)}\\ q &= q^{(1)} = q^{(2)}\\ \pi &= \pi^{(1)} = \pi^{(2)}\\ \mathcal{B} &= \mathcal{B}^{(1)} = \mathcal{B}^{(2)}\\ \mathbf{s} &= \mathbf{s}^{(1)} = \mathbf{s}^{(2)}\end{aligned}$$

- The message and carry subspaces have the same power of two size:

$$\mathsf{msg} = \mathsf{carry} = 2^{\kappa} \text{ for some } \kappa \in \mathbb{N}$$

- The number of bits per block is a power of two:

$$\log_2(\mathsf{p}) = \log_2(\mathsf{msg} \cdot \mathsf{carry}) = 2^{\kappa'} \text{ for some } \kappa' \in \mathbb{N}^*$$

- The carry subspace is assumed to be empty for any input. Using the degree notion (cf. Definition 10):

$$\deg(\mathsf{ct}) = \max_{i \in [0, \mathcal{B}-1]} \{\deg(\mathsf{ct}_i)\} < \mathsf{msg}$$

- At any point of an algorithm, the degree of a ciphertext never exceeds the maximal degree value:

$$\deg(\mathsf{ct}) = \max_{i \in [0, \mathcal{B}-1]} \{\deg(\mathsf{ct}_i)\} < \mathsf{msg} \cdot \mathsf{carry}$$

- At any point in an algorithm, to preserve correctness, no ciphertext block is used as an operand of a dot product with a 2-norm larger than $\nu = \left\lfloor \frac{\mathsf{p}-1}{\mathsf{msg}-1} \right\rfloor$ (see Theorem 4).

The last point could be needed in the case where the ciphertext encrypts a Boolean value. In this, the theoretical number of possible additions might be larger by only taking the degree into account. By verifying all these constraints, we ensure the correctness of all our algorithms with a failure probability smaller than p_{fail} (as defined in Theorem 12). If any exception is made on one of these assumptions, it will be explicitly stated in the respective algorithm.

3.2.3 Toolbox

We also use a variety of standard functions directly in our homomorphic integer algorithms, in particular:

Generic Functions

- $X.\mathsf{append}(y)$: the append method which operates on an existing list X and adds a new element y to the end of X. Can also be used to append one list to another.
- $\mathsf{len}(X)$, $X.\mathsf{len}()$: the length function which outputs the number of elements in a list X

- $X.\mathsf{remove}(i)$: removes the i^{th} element from a list X , and shifts all elements with index $> i$ to the left by one.
- $X.\mathsf{pop}()$: removes the last element from a list and returns it.
- $X \,//\, Y$: quotient of integer division.
- $X \ll y$: shifts bits of X to the left by y spaces.
- $X \gg y$: shifts bits of X to the right by y spaces.
- $\mathsf{isSigned}(\overrightarrow{\mathsf{ct}})$: returns **true** if the homomorphic integer ($\overrightarrow{\mathsf{ct}}$) represents a signed integer type or **false** if it represents an unsigned integer.

Algorithm 23: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathsf{BitExtraction}\Big(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}, \mathsf{offset}, \mathsf{numBits}\Big)$

Input:
- $\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$: a homomorphic integer
- $\mathsf{offset} \in [0, \log_2(\mathsf{msg}) - 1]$: bit index at which the extracted bit shall be placed in the output
- $\mathsf{numBits} \in [0, \log_2(\Omega) - 1]$: number of bits from $\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$ to be extracted

Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

for $i \in [0, \mathsf{numBits} - 1]$ **do**
  $\mathsf{blockIndex} \leftarrow i \,//\, \log_2(\mathsf{msg})$
  $\mathsf{bitIndex} \leftarrow i \bmod \log_2(\mathsf{msg})$
  $\mathsf{lut} \leftarrow \mathbf{GenLUT}\Big((x) \mapsto ((x \gg \mathsf{bitIndex}) \,\&\, 1) \ll \mathsf{offset}\Big)$
  $\mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{PBS}\Big(\mathsf{lut}, \mathsf{ct}_{\mathsf{blockIndex}}^{(\mathsf{in})}\Big)$
return $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

Look-Up Table Generation Functions. In some cases, the generation of the LUT is done beforehand. Taking as input any function from $\mathsf{f} : \mathbb{Z}_\mathsf{p} \to \mathbb{Z}_\mathsf{p}$ (or any subspaces in $\mathbb{Z}_\mathsf{p}$), the algorithm **GenLUT** outputs a well-formed $\mathsf{LUT} \in \mathsf{GLWE}_{\mathbf{S}'}\Big(\widetilde{M_f}\Big) \in \mathfrak{R}_{q,N}^{k+1}$ following Definition 13. We also abuse the notation, by simply denoting this lookup table as LUT_f with

$$\mathsf{LUT}_\mathsf{f} \leftarrow \mathbf{GenLUT}\big(\mathsf{f}\colon (\mathsf{x}) \mapsto \mathsf{f}(\mathsf{x})\big).$$

Similarly in the case of bivariate functions $f(x, y)$, we apply Remark 28, which leads to the following signature:

$$\mathsf{LUT}_\mathsf{f} \leftarrow \mathbf{GenLUT}\big(\mathsf{f}\colon (\mathsf{x}, \mathsf{y}) \mapsto \mathsf{f}(\mathsf{x}, \mathsf{y})\big).$$

PBS Functions. Let us start by recalling the PBS signatures:

- From Algorithm 19, the usual PBS is denoted by $\mathsf{ct}_\mathsf{out} \leftarrow \mathsf{PBS}(\mathsf{CT}_f, \mathsf{ct}_\mathsf{in}, \mathsf{BSK}, \mathsf{DriftMitigation})$;
- From Remark 28, $\mathsf{ct}_\mathsf{out} \leftarrow \mathsf{BivPBS}\big(\mathsf{CT}_{f(x,y)}, \mathsf{ct}_{\mathsf{in}_1}, \mathsf{ct}_{\mathsf{in}_0}, \mathsf{BSK}, \mathsf{DriftMitigation}\big)$ is the signature of the bivariate PBS.

In what follows, we simplify these: the PBS now takes as input either the function definition directly or the associated LUT, and one or two ciphertexts. The bootstrapping key BSK and the $\mathsf{DriftMitigation}$ are also abstracted away. In summary, these are the possibilities:

$$\mathsf{PBS}\Big(\mathsf{LUT_f}, \overrightarrow{\mathsf{ct}}^{(1)}\Big) \coloneqq \mathsf{PBS}\Big(\mathsf{LUT_f}, \overrightarrow{\mathsf{ct}}^{(1)}, \mathsf{BSK}, \mathsf{DriftMitigation}, \mathsf{DMK}\Big);$$

$$\mathsf{PBS}\Big(\mathsf{f}\colon (\mathsf{x}) \mapsto \mathsf{f}(\mathsf{x}), \overrightarrow{\mathsf{ct}}^{(1)}\Big) \coloneqq \mathsf{LUT_f} \leftarrow \mathbf{GenLUT}\big(\mathsf{f}\colon (\mathsf{x}) \mapsto \mathsf{f}(\mathsf{x})\big), \\ \mathsf{PBS}\Big(\mathsf{LUT_f}, \overrightarrow{\mathsf{ct}}^{(1)}, \mathsf{BSK}, \mathsf{DriftMitigation}, \mathsf{DMK}\Big);$$

$$\mathsf{BivPBS}\Big(\mathsf{LUT_f}, \overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big) \coloneqq \mathsf{BivPBS}\Big(\mathsf{LUT_f}, \overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}, \mathsf{BSK}, \mathsf{DriftMitigation}, \mathsf{DMK}\Big);$$

$$\mathsf{BivPBS}\Big(\mathsf{f}\colon (\mathsf{x},\mathsf{y}) \mapsto \mathsf{f}(\mathsf{x},\mathsf{y}), \overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big) \coloneqq \mathsf{LUT_f} \leftarrow \mathbf{GenLUT}\big(\mathsf{f}\colon (\mathsf{x},\mathsf{y}) \mapsto \mathsf{f}(\mathsf{x},\mathsf{y})\big), \\ \mathsf{BivPBS}\Big(\mathsf{LUT_f}, \overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}, \mathsf{BSK}, \mathsf{DriftMitigation}, \mathsf{DMK}\Big).$$

3.3 Bitwise Operations

In this subsection, we outline bitwise operations (such as $\mathsf{AND}, \mathsf{OR}, \mathsf{XOR}$) for cases where both operands are ciphertexts and where one operand is a ciphertext and the other a plaintext. We also consider NOT in the encrypted case.

Encodings. Bitwise operations can be done in both *arithmetic* and *logical* mode, i.e., using the original carry-message encoding (with empty carry bits, see Section 2.3.1), or the Boolean encoding (usually just a single block, see Section 2.3.2). By default, we assume the logical mode (e.g., for Boolean flags), however, we put the algorithms using the generic notation.

3.3.1 Ciphertext-Ciphertext Addition

Bitwise operations are easy to implement as long as msg is a power of 2 and the use of a bivariate PBS is possible. Using the bivariate PBS allows for computing the bitwise operation in a block-by-block manner since there are no dependencies across blocks like there would be with an arithmetic operation. We present a generic bitwise algorithm for $\mathsf{AND}, \mathsf{OR}, \mathsf{XOR}$ in Algorithm 24 where we use the symbol $\circledast$ to denote any two-input Boolean operator. The bitwise NOT operation can be done without using the PBS, as presented in Algorithm 25.

Algorithm 24: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathsf{Bitwise}\Big(\circledast(\cdot,\cdot), \overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big)$

Input: $\begin{cases} \circledast(\cdot,\cdot) : \text{ a binary bitwise operator} \\ \overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)} : \text{ two homomorphic integers} \end{cases}$

Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$: a homomorphic integer

1 **for** $i \in [0, \mathcal{B}-1]$ **do**

2 $\quad \mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{BivPBS}\Big(\circledast(\cdot,\cdot), \mathsf{ct}_i^{(1)}, \mathsf{ct}_i^{(2)}\Big)$

3 **return** $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

Theorem 13 *Let $\overrightarrow{\mathsf{ct}}^{(1)}$ and $\overrightarrow{\mathsf{ct}}^{(2)}$ be two large homomorphic integers. Let $\circledast$ be a bitwise operator[8]. Then, $\mathsf{Dec_s}\Big(\mathsf{Bitwise}\Big(\circledast(\cdot,\cdot), \overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big)\Big) = m^{(1)} \circledast m^{(2)}$. The algorithmic complexity of Algorithm 24 is:*

$$\mathbb{C}^{\mathsf{Bitwise}} = \mathcal{B} \cdot \mathbb{C}^{\mathsf{PBS}}$$

[8] In TFHE-rs, we have implemented the $\mathsf{OR}, \mathsf{AND}, \mathsf{XOR}$ operators, and these are included in our benchmarks.

Algorithm 25: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathsf{BitwiseNOT}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}\right)$

Input: $\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$: a homomorphic integer
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$: a homomorphic integer
for $i \in [0, \mathcal{B}-1]$ **do**
  $\mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{ct}^{(\mathsf{Triv})}(\mathsf{msg}-1) - \mathsf{ct}_i^{(\mathsf{in})}$
return $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

3.3.2 Plaintext-Ciphertext Addition

When one of the inputs to a bitwise operation is a scalar (so unencrypted) we do not need to use the bivariate PBS. Instead, we can use the usual PBS and construct the LUTs from the scalar.

Algorithm 26: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathsf{Bitwise}\left(\circledast(\cdot,\cdot), \overrightarrow{\mathsf{ct}}^{(\mathsf{in})}, \upsilon\right)$

Input:
- $\circledast(\cdot,\cdot)$: a binary bitwise operator
- $\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$: a homomorphic integer
- $\upsilon \in \mathbb{Z}_\Omega$: a scalar

Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$: a homomorphic integer
for $i \in [0, \mathcal{B}-1]$ **do**
  $\mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{PBS}\left(\circledast(\cdot, \upsilon \bmod \mathsf{msg}), \mathsf{ct}_i^{(\mathsf{in})}\right)$
  $\upsilon \leftarrow \upsilon \,//\, \mathsf{msg}$
return $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

Theorem 14 *Let $\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$ be a large homomorphic integer and $\upsilon \in \mathbb{N}$ be a scalar. Let $\circledast$ be a bitwise operator[8]. Then, $\mathsf{Dec_s}\left(\mathsf{Bitwise}\left(\circledast(\cdot,\cdot), \overrightarrow{\mathsf{ct}}^{(\mathsf{in})}, \upsilon\right)\right) = m^{(\mathsf{in})} \circledast \upsilon$. The algorithmic complexity of Algorithm 26 is:*

$$\mathbb{C}^{\mathsf{Bitwise}} = \mathcal{B} \cdot \mathbb{C}^{\mathsf{PBS}}$$

3.4 Equality

In this subsection, we introduce a method to check equality between two homomorphic integers and between a homomorphic integer and a vector of plaintexts.

3.4.1 Ciphertext-Ciphertext Equality

Equality works by using bivariate PBS on pairs of blocks from both inputs with the LUT defined by the function

$$\mathsf{eq}(x,y) \rightarrow \begin{cases} 1 & \text{if } x = y, \\ 0 & \text{otherwise.} \end{cases}$$

This gives a list of blocks where each encrypted value in $[0,1]$.[9] The final part is to reduce this list to one block that encrypts 1 if all blocks encrypt 1, otherwise 0. Note that algorithms computing the equality Eq and the difference Neq are almost the same. The only changes lie in the functions

[9] As each block only encrypts a Boolean value, theoretically, the number of values we can add together, denoted by Chunksize in the algorithm could be larger. However, we do not leverage this, as performing additional additions could exceed the noise threshold. For simplicity, we avoid this approach.

computed by each PBS. We highlight them inside the algorithm by placing the not-equal code inside a dashed box.

Algorithm 27: $\mathsf{ct}^{(\mathsf{Eq})} \leftarrow \mathsf{Eq}\left(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\right)$ $\mathsf{ct}^{(\mathsf{Neq})} \leftarrow \mathsf{Neq}\left(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\right)$

Input: $\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}$: two homomorphic integers
Output: $\mathsf{ct}^{(\mathsf{Eq})}$: A one block ciphertext, encrypting a Boolean value

$\mathsf{cmp} \leftarrow \{\}$
for $i \in [0, \mathcal{B}-1]$ **do**
  $\mathsf{cmp.append}\left(\mathsf{BivPBS}\left(\mathsf{eq}(\cdot,\cdot), \overrightarrow{\mathsf{ct}}_i^{(1)}, \overrightarrow{\mathsf{ct}}_i^{(2)}\right)\right)$ $\mathsf{cmp.append}\left(\mathsf{BivPBS}\left(\mathsf{!eq}(\cdot,\cdot), \overrightarrow{\mathsf{ct}}_i^{(1)}, \overrightarrow{\mathsf{ct}}_i^{(2)}\right)\right)$
$\mathsf{chunkSize} \leftarrow \left\lfloor \frac{\mathsf{p}-1}{\mathsf{msg}-1} \right\rfloor$; // the maximum number of values we can add together
$\mathsf{tmp} \leftarrow \{\}$
while $\mathrm{len}(\mathsf{cmp}) > 1$ **do**
  $(\mathsf{numFullChunks}, \mathsf{rem}) \leftarrow (\mathrm{len}(\mathsf{cmp}) \,//\, \mathsf{chunkSize}, \mathrm{len}(\mathsf{cmp}) \bmod \mathsf{chunkSize})$
  for $i \in [0, \mathsf{numFullChunks} - 1]$ **do**
    $\mathsf{sum} \leftarrow \sum_{j=0}^{\mathsf{chunkSize}-1} \mathsf{cmp}_{i \cdot \mathsf{chunkSize}+j}$
    $\mathsf{tmp.append}(\mathsf{PBS}(\mathsf{eq}(\cdot, \mathsf{chunkSize}), \mathsf{sum}))$ $\mathsf{tmp.append}(\mathsf{PBS}(\mathsf{!eq}(\cdot, 0), \mathsf{sum}))$
  $\mathsf{sum} \leftarrow \sum_{j=\mathrm{len}(\mathsf{cmp})-\mathsf{rem}}^{\mathrm{len}(\mathsf{cmp})-1} \mathsf{cmp}_j$
  $\mathsf{tmp.append}(\mathsf{PBS}(\mathsf{eq}(\cdot, \mathsf{rem}), \mathsf{sum}))$ $\mathsf{tmp.append}(\mathsf{PBS}(\mathsf{!eq}(\cdot, 0), \mathsf{sum}))$
  $\mathsf{cmp} \leftarrow \mathsf{tmp}$
  $\mathsf{tmp} \leftarrow \{\}$
return cmp

Theorem 15 *Let* $\overrightarrow{\mathsf{ct}}^{(1)}$ *and* $\overrightarrow{\mathsf{ct}}^{(2)}$ *be two large homomorphic integers, and let* $l = \left\lfloor \frac{\mathsf{p}-1}{\mathsf{msg}-1} \right\rfloor$ *be the maximal number of values we can add together. Then,* $\mathsf{Dec_s}\left(\mathsf{Eq}\left(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\right)\right) = \mathsf{eq}(m^{(1)}, m^{(2)}) \in \mathbb{Z}_2$. *The algorithmic complexity of Algorithm 27 is:*

$$\mathbb{C}^{\mathsf{Eq}} = \left(\mathcal{B} + \sum_{i=0}^{\lceil \log_l(\mathcal{B}) \rceil - 1} \left\lceil \frac{\mathcal{B}}{l^{i+1}} \right\rceil\right) \cdot \mathbb{C}^{\mathsf{PBS}}$$

3.4.2 Plaintext-Ciphertext Equality

The equality with a clear scalar uses the same logic as the equality with an encrypted value. One improvement is to pack pairs of block into a single LWE ciphertext using the shift and add technique ($\mathsf{pack}(\mathsf{a}, \mathsf{b}) = (\mathsf{a} \cdot \mathsf{msg}) + \mathsf{b}$), halving the number of blocks in the list that needs to be then reduced. This is possible as we don't need to use the bivariate PBS in this scenario. The complexity of the algorithm becomes:

$$\mathbb{C}^{\mathsf{Eq}} = \left(\left\lceil \frac{\mathcal{B}}{2} \right\rceil + \sum_{i=0}^{\lceil \log_l(\lceil \frac{\mathcal{B}}{2} \rceil) \rceil - 1} \left\lceil \frac{\lceil \frac{\mathcal{B}}{2} \rceil}{l^{i+1}} \right\rceil\right) \cdot \mathbb{C}^{\mathsf{PBS}}$$

3.5 Carry Propagation

During arithmetic operations such as addition, subtraction, negation, or multiplication, the carry buffers in each block gradually fill up. At some point, carry propagation becomes necessary, either

because the buffers are full and the next operation risks overflow, or because the next operation requires an empty carry space.

Propagating carries is a very sequential operation as adding the carry stored in the block n to the block $n+1$ will modify the carry of block $n+1$. The basic implementation of carry propagation using a fully sequential algorithm is given in Algorithm 28. In it, starting from the block corresponding to the least significant digit, we extract the value stored in the message space and the value stored in the carry space using two separate PBSes. We then add the value that was in the carry space to the next block before repeating this process. As such, the carry space in every block (except the first) needs to be able to accommodate one further addition without the carry space overflowing; that is the block can encrypt at most the value $\mathsf{carry} \cdot (\mathsf{msg} - 1)$.

Algorithm 28: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathsf{SequentialCarryPropagation}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}\right)$

Input: $\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$: a homomorphic integer with partially filled carry buffers
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$: a homomorphic integer with empty carry buffers

$\mathsf{ct}^{(\mathsf{carry})} \leftarrow \mathsf{ct}^{(\mathsf{Triv})}(0)$
for $i \in [0, \mathcal{B}-1]$ **do**
  $\mathsf{ct}^{(\mathsf{tmp})} \leftarrow \mathsf{ct}_i^{(\mathsf{in})} + \mathsf{ct}^{(\mathsf{carry})}$
  $\mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{PBS}\big((x) \mapsto x \bmod \mathsf{msg}, \mathsf{ct}^{(\mathsf{tmp})}\big)$
  $\mathsf{ct}^{(\mathsf{carry})} \leftarrow \mathsf{PBS}\big((x) \mapsto x \,//\, \mathsf{msg}, \mathsf{ct}^{(\mathsf{tmp})}\big)$
return $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

The sequential algorithm requires a total of $2\mathcal{B}$ PBSes (or $2\mathcal{B}-1$ if the carry from the last block is not needed). The latency corresponds to the time required to perform all these PBSes sequentially. If the two PBSes in each iteration are executed in parallel, the latency is $\mathcal{B}$ times the latency of a single PBS. In either case, the carry propagation algorithm's latency scales linearly with both the PBS time and $\mathcal{B}$. Due to the high cost of the PBS, this operation remains slow even for relatively small values of $\mathcal{B}$ (e.g., 16).

When latency is a priority, a parallel algorithm can be used instead. In this section, we introduce an algorithm that requires more computing power but is highly parallelizable, thereby reducing the operation's latency.

Overview of specialised parallel implementation of carry propagation For simplicity, we assume that the carry space contains at most one filled bit and that, if the carry is one, the message cannot reach its maximum value. This assumption does not reduce generality, as in the most common use case, carry propagation is performed after summing two values, each at most $\mathsf{msg}-1$. As a result, the values encrypted in the resulting ciphertexts are smaller than $2 \cdot \mathsf{msg} - 2$, satisfying the previous assumptions.

At a high level, the basic idea is to split the list of blocks into distinct groups of adjacent blocks, compute some state information that depends only on blocks within a single group in parallel, compute the carries between groups, and then propagate all this information to all blocks to give the result.

Suppose we want to determine whether block b within group g will receive a carry during carry propagation. Assume we have in encrypted form the carry (so either zero or one) of the previous group $g-1$. We can then determine whether block b of group g will receive a carry based on the so-called group carry bit of group $g-1$ and information about the values in the preceding blocks (i.e., blocks $0, \ldots, b-1$) of group g. Specifically, we need to know whether these preceding blocks generate a carry that propagates to block b or whether a carry from the previous group would propagate to this block. What the algorithm will do is construct a ciphertext that encrypts a value in $\{0, 1, 2\}$, which we call the

propagation state, with the value 2 being if the previous blocks in the group generate and propagate a carry to the block b, 1 if the previous blocks would only propagate a carry from the previous group and 0 otherwise, i.e., if any carry from the previous group or blocks in the current group would be absorbed before reaching this block.

Once we have both the group carry of the previous group and the propagation state of the block, we can compute via a PBS the value $(x \bmod \mathsf{msg}) \ll 1$, where $x \bmod \mathsf{msg}$ is the value encrypted in the message part of block b of group g, and then add the group carry and the propagation state. Here, we observe that if a carry is propagated to this block, the sum of the group carry and the propagation state is either 2 or 3; otherwise, it is 0 or 1. Shifting the result of this sum to the right by one bit produces a single bit that is 1 if a carry is propagated to this block (i.e., if the sum was either 2 or 3), and 0 otherwise (i.e., if the sum was either 0 or 1). Note that this technique remains valid even if the carry space consists of a single bit.

The algorithm consists of several steps, each of which is explained in further detail in the following subsections. Below, we provide a simplified overview of each step:

1. First, blocks are grouped, and for each block (in parallel), a *block state* is computed along with a cleaned version of the block (i.e., with the carry space emptied), shifted left by one bit.
2. Next, for each group (in parallel), an internal propagation of state information is performed to compute the *propagation states* of each block, as well as unresolved information about the *group carries*.
3. Then, the carry bit between groups is determined from the unresolved group carries. We provide two methods for this: one sequential and one using the Hillis-Steele scan, a parallel scan algorithm.
4. Finally, in parallel, the group carries and propagation states are added to the cleaned and shifted blocks, and the resulting blocks are shifted down by one bit to produce the final result.

3.5.1 Step 1: Groups and Block States

Blocks are grouped in groups of $\log_2(\mathsf{p})$ elements (so with the standard $\mathsf{p} = 16$, each group consists of 4 blocks). For each block, we will compute a so-called *block state* which encodes information about the value it encrypts. For each block, we will first associate it with an encoded value in $\{0, 1, 2\}$ depending on whether the block

- generates a carry,
- may propagate the carry coming from the previous block,
- does neither, i.e., will absorb any input carry.

The encoding is 2 for *Generate*, 1 for *Propagate* and 0 for *Neither*. The way to compute this encoding is by homomorphically comparing the encrypted value x with msg using a PBS by applying the function

$$\mathsf{unshifted\_state}(x) = \begin{cases} 2, & \text{if } x \geq \mathsf{msg}; \\ 1, & \text{if } x = \mathsf{msg} - 1; \\ 0, & \text{otherwise.} \end{cases}$$

This works because we have restricted the carry to be a single bit so it can only be propagated if it is added to a value that is at least $\mathsf{msg} - 1$. Furthermore, a block cannot simultaneously generate a

carry and propagate a carry since then it would encrypt a value bigger than $2 \cdot \mathsf{msg} - 1$ which would break our assumption on the maximum value that a block can encrypt.

Instead of applying unshifted_state to each block, we will actually apply a shifted version where we left-shift by the index i used to index its position within its group, with the exception that for blocks in the first group, the left-shift used is $i - 1$. This shift is required so we can propagate these states in the next step. Note that the first block of the first group is "left-shifted" by -1 bits by which we mean right shift by 1 bit, i.e., dropping the least significant bit. This does not lose information since the very first block does not receive any carry so it cannot propagate. Written another way, the very first block of the homomorphic integer has a state that is given by the Boolean function $x \geq \mathsf{msg}$. In summary, we use the following functions to determine the state of a given block, where i is the index of the block within its group:

State function for blocks in the first group:

$$\begin{cases} 2 \ll (i-1), & \text{if } x \geq \mathsf{msg}, \\ 1 \ll (i-1), & \text{if } x = \mathsf{msg} - 1, \\ 0, & \text{otherwise.} \end{cases} \tag{30}$$

State function for all other blocks:

$$\begin{cases} 2 \ll i, & \text{if } x \geq \mathsf{msg}, \\ 1 \ll i, & \text{if } x = \mathsf{msg} - 1, \\ 0, & \text{otherwise.} \end{cases} \tag{31}$$

We note that all these functions take the same form and depend only on the shift used. Further, we point out that if we have a propagate state in the last block of a group then the state information will occupy the padding bit. This is not a problem and will be dealt with later.

Finally, there is one further exception to how we compute the block states. The block state of the very last block is only relevant if we want to check if an unsigned overflow happened as it would only be used to compute the group carry of the last group. Since we do not need this group carry to compute the final result, we avoid computing it as it can affect the latency of the next steps. This is the reason for this exception. Instead, for the last block, we will shift the unshifted state by either 1 bit to the left if the last block is the only block in its group and otherwise by 2 bits to the left if there are other blocks in the same group. More details will be given later in Section 3.14.

Algorithm 30 describes how to compute all block states and how to clean and shift each block. We leverage the PBS to compute all block states using the lookup tables defined in Algorithm 29 and to generate the cleaned-shifted version of each block, this time using the function $\mathsf{cleanAndShift}(x) = (x \bmod \mathsf{msg}) \ll 1$.

For parameters with $\mathsf{p} \geq 16$ which have at least 4 bits in the combined message and carry space, the manyLUT technique (see Algorithm 21) can be used, as only 3 out of $\log_2(\mathsf{p})$ bits are utilized. Leveraging the manyLUT PBS reduces the number of PBS operations by half.

Algorithm 29: Block States Lookup Tables

Function `CreateBlockStatesLuts(groupSize)`:
  Input: $\mathsf{groupSize} \in \mathbb{N}$
  Output: blockStatesLuts : list of $\mathsf{groupSize}+1$ lookup tables
  $\mathsf{blockStatesLuts} \leftarrow \left\{\mathbf{GenLUT}\Big((x) \mapsto x \geq \mathsf{msg}\Big)\right\}$
  for $i \in [0, \mathsf{groupSize}-1]$ **do**
    Function `LocalFn`(x):
      if $x \geq \mathsf{msg}$ **then**
        return $2 \ll i$
      else if $x = \mathsf{msg}-1$ **then**
        return $1 \ll i$
      else
        return $0 \ll i$
    $\mathsf{blockStatesLuts.append}\big(\mathbf{GenLUT}\big(\texttt{LocalFn}(\cdot)\big)\big)$
  return blockStatesLuts

Algorithm 30: Shifted block and block states computation

Function `ComputeShiftedBlocksAndStates`$(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})})$:
  Input: $\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$: a homomorphic integer with potentially non-empty carry spaces
  Output: $\begin{cases}\overrightarrow{\mathsf{ct}}^{(\mathsf{shiftedBlocks})} \\ \overrightarrow{\mathsf{ct}}^{(\mathsf{blockStates})}\end{cases}$: two homomorphic integers (the integer encrypted has no real meaning)
  $\mathsf{groupSize} \leftarrow \lfloor \log_2(\mathsf{p}) \rfloor$
  $\mathsf{blockStatesLuts} \leftarrow$ `CreateBlockStatesLuts`(groupSize)
  /* `This can be done in parallel` */
  for $i \in [0, \mathcal{B}-1]$ **do**
    $\mathsf{groupIndex} \leftarrow i \;//\; \mathsf{groupSize}$
    $\mathsf{indexInGroup} \leftarrow i \bmod \mathsf{groupSize}$
    if $i = \mathcal{B}-1$ **then**
      $\mathsf{lut} \leftarrow \mathsf{blockStatesLuts}[3]$
    else if $\mathsf{groupIndex} = 0$ **then**
      $\mathsf{lut} \leftarrow \mathsf{blockStatesLuts}[\mathsf{indexInGroup}]$
    else
      $\mathsf{lut} \leftarrow \mathsf{blockStatesLuts}[\mathsf{indexInGroup}+1]$
    $\mathsf{ct}^{(\mathsf{blockStates})}[i] \leftarrow \mathsf{PBS}\Big(\mathsf{lut}, \overrightarrow{\mathsf{ct}}_i^{(\mathsf{in})}\Big)$
    $\mathsf{ct}^{(\mathsf{shiftedBlocks})}[i] \leftarrow \mathsf{PBS}\Big((x) \mapsto (x \bmod \mathsf{msg}) \ll 1, \overrightarrow{\mathsf{ct}}_i^{(\mathsf{in})}\Big)$
  return $\overrightarrow{\mathsf{ct}}^{(\mathsf{shiftedBlocks})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{blockStates})}$

3.5.2 Step 2: Propagation States and Group Carries

The next step is to update the block state information to form the *propagation states* and compute the so-called *group carries* for each group, the idea of the group carries is the same as the block state for a block but at the group level. Does the group generate a carry, if not, does it propagate a carry from the previous group, or does it absorb any carry that might come from a previous group? Again, the first group is special as it cannot propagate since it receives no carry.

Clearly, if the final block in a group generates a carry, so does the group itself and more generally, the group will generate one if one of its blocks does and then all subsequent blocks in the group propagate. Further, the group propagates if and only if all its blocks propagate. Any other combination of block states is able to absorb a carry within the group since the carry is a single bit.

With this information, we can see that if we add all the (shifted) block states of the blocks in a group (excluding the first group), then the group state will be 'generate' if the sum is at least p (i.e., the padding bit is one), 'propagate' if the sum is $\mathsf{p}-1$, and otherwise will be 'neither' (any carry is absorbed).

An example of all three cases when a group consists of two blocks is shown below. Recall that block states are shifted based on their index within the group.

2 Propagate:

$$\begin{array}{rr} & 001 \\ + & 010 \\ \hline \mathsf{Propagate} & 011 \end{array}$$

1 Generate, 1 Propagate:

$$\begin{array}{rr} & 010 \\ + & 010 \\ \hline \mathsf{Generate} & 100 \end{array}$$

1 Generate, 1 Neither:

$$\begin{array}{rr} & 010 \\ + & 000 \\ \hline \mathsf{Neither} & 010 \end{array}$$

For the first group, again there is no input carry so either the group generates a carry or it does not. As we shifted by one less bit, when summing, we have that the group 'generates' if the sum is at least $\frac{\mathsf{p}}{2}$. Thus for the first group, we have a resolved group carry and for the other groups, we have only unresolved group carries.

Clearly, this technique is applicable to any of the blocks in a group, not just the last one, so in the same manner, we will use the block states to compute the *propagation state* of each block to determine whether a carry (either from a previous block in the group or propagation of a carry from the previous group) reaches that block. All we have to do is replace p in the above by 2^{i+1} where i is the index of the block in the group.

Thus, for each block that is not the first in a group, a new propagation state is computed using the function given in Eq. (33) applied to the sum of the block states up to but not including that block in a given group. For the first block in a group, only the group carry from the previous group matters, in this case, we define the propagation state to be a trivial encryption of 1, thus when we later add the resolved group carry for the previous group to it, it will be 2 or 1 in line with what is required.

Again, for the blocks of the first group, the function is slightly different. As already pointed out, the first block state is already resolved, thus the propagation states of the blocks in the first group will be resolved at this step to either 2 or 0, depending on if a carry needs to be added to this block or not.

Propagation state function for the first group applied to the partial sum of the first i blocks (excluding $i = \log_2(\mathsf{p})$):

$$\begin{cases} 2, & \text{if } x \geq 1 \ll i, \\ 0, & \text{otherwise.} \end{cases} \tag{32}$$

Propagation state function for all other groups applied to the partial sum of the first i blocks (excluding $i = \log_2(\mathsf{p})$):

$$\begin{cases} 2, & \text{if } x \geq 2 \ll i, \\ 1, & \text{if } x = (2 \ll i) - 1, \\ 0, & \text{otherwise.} \end{cases} \tag{33}$$

The group carry state of a group (excluding the first group) cannot be computed in this way due to the fact that the padding bit may not be zero (since we are in the case $i = \log_2(\mathsf{p})$). Furthermore, we will see in the next step that we will want to use a different function depending on which of the two methods of resolving the group carry state is used.

The trick for dealing with a padding bit, which may be non-zero, is to note that if the padding bit is one, then the resulting PBS will output an encryption of the negated value of the PBS where the padding bit has zero instead. By being clever with the function we evaluate and adding a well-chosen constant to the result, we will be able to differentiate the three cases. Something we now explain for the two approaches.

In the case of the sequential algorithm, we want to use the usual encoding in $\{0, 1, 2\}$ shifted by a certain number of bits. To achieve this, we can first apply the function

$$\mathsf{sequential\_state}(x) = \begin{cases} 0, & \text{if } x = \mathsf{p} - 1, \\ -1 \ll (\mathsf{groupIndex} \bmod (\log_2(\mathsf{p}) - 1)), & \text{otherwise,} \end{cases}$$

and then add the constant $1 \ll (\mathsf{groupIndex} \bmod (\log_2(\mathsf{p}) - 1))$. Here, $\mathsf{groupIndex}$ is the index of the group in the list of all groups whose group carry state is unresolved (i.e., all groups except the first – so the second group has index 0, the third group has index 1, etc.). An example for $\mathsf{groupIndex} = 0$ illustrating why this works is given below.

- For $x \in [0, \mathsf{p} - 2]$, the result of this operation will be $\mathsf{sequential\_state}(x) + 1 = -1 + 1 = 0$ which means neither propagate nor generate.
- For $x = \mathsf{p} - 1$, the result will be $\mathsf{sequential\_state}(x) + 1 = 0 + 1$ which means propagate state.
- For $x \in [\mathsf{p}, 2\mathsf{p} - 2]$, the result will be $\mathsf{sequential\_state}(x) = -\mathsf{sequential\_state}(x - \mathsf{p}) + 1 = -(-1) + 1 = 2$ which means generate, so all cases are correct.

We remark that x cannot be $2\mathsf{p} - 1$ since this value cannot be reached by the sum of the block states from step one.

In the case of the Hillis-Steele prefix scan method, we will change the encoding slightly and instead apply the function

$$\mathsf{HS\_state}(x) = \begin{cases} 2, & \text{if } x = \mathsf{p} - 1, \\ -1, & \text{otherwise,} \end{cases}$$

and then add the clear constant 1 to the result. In this case, the resulting encoding is 2 for generate, 3 for propagate, and 0 for neither. Note that we do not associate 1 with any state due to a restriction in the PBS applying $\mathsf{HS\_state}$. However, we will later associate 1 with a generate state as well.

Finally, the case of the group carry for the first group does not have the same issue with the padding bit and we can directly apply a PBS with the Boolean function $x \geq \frac{\mathsf{p}}{2}$ to give the resolved group carry (we do not shift by 1 bit here as we want a value in $\{0, 1\}$).

The algorithm to construct all the necessary lookup tables in this step is given in Algorithm 31. Computation of the propagation states and group carries is given in Algorithm 32. Each lookup table is applied using a PBS and the output ciphertexts corresponding to propagation states will be added to the appropriate block of $\overrightarrow{\mathsf{ct}}^{(\mathsf{shiftedBlocks})}$ from Algorithm 30, these are called *prepared blocks*. Note that if the carry space consists of a single bit, then there is a chance that the padding bit of a prepared block is flipped to one, this will not cause problems later however.

Algorithm 31: LUTs for computing group carries and propagation states

Function `PrepareGroupLuts(groupSize, useSeq)`:

Input: $\begin{cases} \mathsf{groupSize} \in \mathbb{N} \\ \mathsf{useSeq}: \text{ a Boolean} \end{cases}$

Output: $\begin{cases} \mathsf{firstGroupPropStateLuts} \\ \mathsf{otherGroupPropStateLuts} \\ \mathsf{firstGroupCarryLut} \\ \mathsf{otherGroupCarryLuts} \end{cases}$

$\mathsf{firstGroupPropStateLuts} \leftarrow \{\}$
for $i \in [0, \mathsf{groupSize} - 2]$ **do**
  $\mathsf{firstGroupPropStateLuts.append}\Big(\mathbf{GenLUT}\Big((x) \mapsto ((x \gg i)\ \&\ 1) \cdot 2\Big)\Big)$
$\mathsf{otherGroupPropStateLuts} \leftarrow \{\}$
for $i \in [0, \mathsf{groupSize} - 1]$ **do**
  Function `LocalFn`(x)**:**
    if $x \geq (2 \ll i)$ **then**
      $r \leftarrow 2$
    else if $x = (2 \ll i) - 1$ **then**
      $r \leftarrow 1$
    else
      $r \leftarrow 0$
    return r
  $\mathsf{otherGroupPropStateLuts.append}\big(\mathbf{GenLUT}\big(\texttt{LocalFn}(\cdot)\big)\big)$
$\mathsf{firstGroupCarryLut} \leftarrow \mathbf{GenLUT}\Big((x) \mapsto (x \gg (\mathsf{groupSize} - 1))\ \&\ 1\Big)$
$\mathsf{otherGroupCarryLuts} \leftarrow \{\}$
if useSeq **then**
  for $i \in [0, \mathsf{groupSize} - 2]$ **do**
    Function `LocalFn`(x)**:**
      if $x = \mathsf{p} - 1$ **then**
        $r \leftarrow 0$
      else
        $r \leftarrow (-1 \ll i) \bmod (2\mathsf{p})$
      return r
    $\mathsf{otherGroupCarryLuts.append}\big(\mathbf{GenLUT}\big(\texttt{LocalFn}(\cdot)\big)\big)$
else
  Function `LocalFn`(x)**:**
    if $x = \mathsf{p} - 1$ **then**
      $r \leftarrow 2$
    else
      $r \leftarrow -1 \bmod (2\mathsf{p})$
    return r
  $\mathsf{otherGroupCarryLuts.append}\big(\mathbf{GenLUT}\big(\texttt{LocalFn}(\cdot)\big)\big)$
return $\mathsf{firstGroupPropStateLuts}, \mathsf{otherGroupPropStateLuts}, \mathsf{firstGroupCarryLut}, \mathsf{otherGroupCarryLuts}$

Algorithm 32: Compute Propagation States And Group Carries

Function `ComputePropagationStatesAndGroupCarries(`groupSize, $\overrightarrow{\mathsf{ct}}^{(\mathsf{blockStates})}$, useSeq`):`

Input: $\begin{cases} \mathsf{groupSize} \in \mathbb{N} \\ \overrightarrow{\mathsf{ct}}^{(\mathsf{blockStates})} : \text{ a homomorphic integer (the integer encrypted has no real meaning)} \\ \mathsf{useSeq};\ \text{a Boolean} \end{cases}$

Output: $\begin{cases} \overrightarrow{\mathsf{ct}}^{(\mathsf{propStates})} : \text{ a homomorphic integer (the integer encrypted has no real meaning)} \\ \mathsf{groupCarries} : \text{ a list of single block (LWE) ciphetexts} \end{cases}$

(firstGroupPropStateLuts, otherGroupPropStateLuts, firstGroupCarryLut, otherGroupCarryLuts) ← `PrepareGroupLuts(`groupSize, useSeq`)`

sums ← {}

numberofGroups ← $\left\lceil \frac{\mathcal{B}-1}{\mathsf{groupSize}} \right\rceil$

`/* This should be done in parallel */`

for $i \in [0, \mathsf{numberofGroups} - 1]$ **do**

  sums.append$\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{blockStates})}_{i \cdot \mathsf{groupSize}}\right)$

  `/* This must be sequential */`

  for $j \in [1, \mathsf{groupSize} - 1]$ **do**

    if $(i \cdot \mathsf{groupSize}) + j < \mathcal{B} - 1$ **then**

      sums.append$\left(\mathsf{sums.last}() + \overrightarrow{\mathsf{ct}}^{(\mathsf{blockStates})}_{(i \cdot \mathsf{groupSize})+j}\right)$

`/* This should be done in parallel */`

for $i \in [0, \mathcal{B} - 2]$ **do**

  groupIndex ← $\left\lfloor \frac{i}{\mathsf{groupSize}} \right\rfloor$

  indexInGroup ← $i \bmod \mathsf{groupSize}$

  if groupIndex $= 0$ **then**

    if indexInGroup $=$ groupSize $- 1$ **then**

      lut ← firstGroupCarryLut

    else

      lut ← firstGroupPropStateLuts[indexInGroup]

  else if indexInGroup $=$ groupSize $- 1$ **then**

    if useSeq **then**

      lut ← otherGroupCarryLuts[(groupIndex $- 1$) $\bmod$ (groupSize $- 1$)]

    else

      lut ← $\mathsf{otherGroupCarryLuts}_0$

  else

    lut ← otherGroupPropStateLuts[indexInGroup]

  sums[i] ← PBS(lut, sums[i])

  if groupIndex $\neq 0$ **and** indexInGroup $=$ groupSize $- 1$ **then**

    if useSeq **then**

      sums[i] ← sums[i] + $\mathsf{ct}^{(\mathsf{Triv})}(1 \ll (\mathsf{groupIndex} - 1) \bmod (\mathsf{groupSize} - 1))$

    else

      sums[i] ← sums[i] + $\mathsf{ct}^{(\mathsf{Triv})}(1)$

$\overrightarrow{\mathsf{ct}}^{(\mathsf{propStates})}_0 \leftarrow \mathsf{ct}^{(\mathsf{Triv})}(0)$

groupCarries ← {}

`/* Relabeling Step */`

for $i \in [0, \mathcal{B} - 2]$ **do**

  if $i \bmod \mathsf{groupSize} = \mathsf{groupSize} - 1$ **then**

    groupCarries.append(sums[i])

    $\overrightarrow{\mathsf{ct}}^{(\mathsf{propStates})}_{i+1} \leftarrow \mathsf{ct}^{(\mathsf{Triv})}(1)$

  else

    $\overrightarrow{\mathsf{ct}}^{(\mathsf{propStates})}_{i+1} \leftarrow \mathsf{sums}[i]$

return $\overrightarrow{\mathsf{ct}}^{(\mathsf{propStates})}$, groupCarries

3.5.3 Step 3: Resolving the Group Carries

In this step, we want to resolve the group carries to take values in $\{0, 1\}$, depending on whether there is a carry from that group or not. As previously mentioned, we propose two possible algorithms to resolve the group carries:

- A sequential one that resolves $\log_2(\mathsf{p}) - 1$ carries at each iteration, it uses a logic similar to the construction of the propagation states of the first group described earlier;
- A parallel algorithm based on Hillis-Steele prefix scan [HSJ86] which can be applied when $\mathsf{p} \geq 16$.

The sequential algorithm is used for a smaller number of groups until Hillis-Steele becomes the faster option, as determined later in Algorithm 36 using the useSeq flag.

We again note that we are not computing or resolving a group carry for the last group as this may slow down this step. Instead, if needed we can compute whether an overflow occurred later as will be described in Section 3.14.

Sequential Group Carry Propagation. This propagation works in essentially the same way the propagation states were computed for the first group. We process $\log_2(\mathsf{p}) - 1$ unresolved carries at a time, compute their partial sums plus the resolved carry from the previous group, and check if this result is larger than the appropriate power of two to resolve the group carry. Details are given in Algorithm 33.

Algorithm 33: Sequential Group Carry Propagation

```
1  Function ResolveGroupCarriesSequentially(groupCarries, groupSize):
       Input: { groupCarries : a list of single block (LWE) ciphertexts
              { groupSize ∈ ℕ
       Output: resolvedCarries : a list of single block (LWE) ciphertexts
2      luts ← {}
3      for i ∈ [0, groupSize − 1] do
4          luts.append(GenLUT((x) ↦ x ≫ (i + 1) & 1))
5      resolvedCarries ← {}
6      resolvedCarries.append(groupCarries[0])
7      j ← 1
8      lastIndex ← log_2(p) − 2
9      while j < groupCarries.len() do
10         array ← {}
11         array.append(resolvedCarries.last() + groupCarries[j])
12         j ← j + 1
           /* This needs to be done sequentially                          */
13         for i ∈ [1, log_2(p) − 2] do
14             if j < groupCarries.len() then
15                 array.append(array.last() + groupCarries[j])
16                 j ← j + 1
17             else
18                 lastIndex ← i − 1
19                 break
           /* This can be done in parallel                                */
20         for i ∈ [0, lastIndex] do
21             resolvedCarries.append(PBS(luts[i], array[i]))
22     return resolvedCarries
```

Hillis-Steele Group Carry Propagation. Here, we use the prefix scan parallel algorithm, however, instead of doing an addition, we do a bivariate PBS with the function described in Algorithm 34.

Algorithm 34: Lookup Table Function For Hillis-Steele Group Carry Propagation

```
Function HillisSteeleAdd(previous, current):
    Input: previous ∈ {0, 1, 2, 3}
           current ∈ {0, 1, 2, 3}
    Output: result ∈ {0, 1, 2, 3}
    if current = 2 then
        return 1
    else if current = 3 then
        if previous = 2 then
            return 1
        else
            return previous
    else
        return current
```

Recall that for this method, we encoded our unresolved group carries as 3 for propagate, 1 or 2 for generate, and 0 for neither. Thus, we can write this function in the following manner to make it clearer:

previous →		N	G	G	P
↓ current		0	1	2	3
N	0	0	0	0	0
G	1	1	1	1	1
G	2	1	1	1	1
P	3	0	1	1	3

Here, we see that we can only get as output a propagate state if the two inputs are also both propagate. Otherwise, we are in a generate state unless the current state is Neither or the current state is Propagate and the previous state was Neither.

By the end of the Hillis-Steele scan, all Generate states are going to be changed to 1 as it is the value representing that a carry occurs. The Propagate states will also all be resolved to 0 or 1 since the first group carry is already resolved, and this will get combined (via the bivariate PBS) with all states eventually.

Algorithm 35: Hillis Steele Carry Propagation

```
Function ResolveGroupsCarriesUsingHillisSteele(groupCarries):
    Input: groupCarries : a list of single block (LWE) ciphertexts
    Output: resolvedCarries : a list of single block (LWE) ciphertexts
    lut ← GenLUT(HillisSteeleAdd(·, ·))
    /* This needs to be done sequentially                          */
    for d ∈ [0, ⌈log2(groupCarries.len())⌉ − 1] do
        /* This can be done in parallel                            */
        for i ∈ [2^d, groupCarries.len() − 1] do
            groupCarries[i] ← BivPBS(lut, groupCarries[i − 2^d], groupCarries[i])
    resolvedCarries ← groupCarries
    return resolvedCarries
```

3.5.4 Step 4: Final Step

At this step, we have a list of *prepared blocks* and the resolved group carries of each group. The last step consists in adding the resolved group carry of group i to all the prepared blocks of the group $i+1$,

and then using a PBS with the function $(x \gg 1) \bmod \mathsf{msg}$ to finalize the block. Note that for the first group, there is no group carry to add (so that the group carry to be added is a trivial encryption of zero).

Further, in the case of a single bit for the carry space, we can have values up to $2 \cdot \mathsf{msg} + 1$ encrypted after the sum which means the padding bit can be one. However, if the padding bit is one, then the message and carry space contains the value 0 or 1, and the shift function results in the value zero meaning that the negation applied due to the padding bit being one has no effect and there isn't any issue here.

This last step can be seen in Algorithm 36 which puts together Algorithm 30, Algorithm 32 and either Algorithm 33 or Algorithm 35, to show to propagate the carries when they can occupy at most one bit of the carry space.

Algorithm 36: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathsf{PropagateSingleBitCarries}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}\right)$

Input: $\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$: a homomorphic integer with each block encrypting a value in $[0, 2 \cdot \mathsf{msg} - 2]$
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$: a homomorphic integer with empty carry spaces

$\mathsf{groupSize} \leftarrow \lfloor \log_2(\mathsf{p}) \rfloor$
$\mathsf{numCarries} \leftarrow \left\lceil \frac{\mathcal{B}}{\mathsf{groupSize}} \right\rceil - 1$
$\mathsf{seqDepth} \leftarrow \left\lfloor \frac{\mathsf{numCarries}-1}{\mathsf{groupSize}-1} \right\rfloor$
$\mathsf{hillisSteeleDepth} \leftarrow$ **if** $\mathsf{numCarries} = 0$ **then** 0 **else** $\lceil \log_2(\mathsf{numCarries}) \rceil$
$\mathsf{useSeq} \leftarrow \mathsf{seqDepth} \leq \mathsf{hillisSteeleDepth}$ or $\mathsf{p} < 16$
$\overrightarrow{\mathsf{ct}}^{(\mathsf{shiftedBlocks})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{blockStates})} \leftarrow \mathtt{ComputeShiftedBlocksAndStates}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}\right)$
$\overrightarrow{\mathsf{ct}}^{(\mathsf{propStates})}, \mathsf{groupCarries} \leftarrow$
  $\mathtt{ComputePropagationStatesAndGroupCarries}\left(\mathsf{groupSize}, \overrightarrow{\mathsf{ct}}^{(\mathsf{blockStates})}, \mathsf{useSeq}\right)$
if $\mathsf{numCarries} = 0$ **then**
  $\mathsf{resolvedCarries} \leftarrow \left\{\mathsf{ct}^{(\mathsf{Triv})}(0)\right\}$
else if useSeq **then**
  $\mathsf{resolvedCarries} \leftarrow \mathtt{ResolveGroupCarriesSequentially}(\mathsf{groupCarries}, \mathsf{groupSize})$
else
  $\mathsf{resolvedCarries} \leftarrow \mathtt{ResolveGroupsCarriesUsingHillisSteele}(\mathsf{groupCarries})$
/* This can be done in parallel */
for $i \in [0, \mathcal{B} - 1]$ **do**
  $\mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{ct}_i^{(\mathsf{shiftedBlocks})} + \mathsf{ct}_i^{(\mathsf{propStates})} + \mathsf{resolvedCarries}\left[\left\lfloor \frac{i}{\mathsf{groupSize}} \right\rfloor\right]$
  $\mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{PBS}\left((x) \mapsto (x \gg 1) \bmod \mathsf{msg}, \mathsf{ct}_i^{(\mathsf{out})}\right)$
return $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

3.5.5 Full-Fledged Carry Propagation

With the ability to propagate single-bit carries in a parallel manner, we can now give the final algorithm which shows how to use this to propagate any size carries in parallel. This can be found in Algorithm 37. The important observation is that when adding the carry space to the next message space, there is at most one bit of further carry created, it is this carry that can then be propagated using the previous algorithm. We can also avoid any unnecessary carry propagation by only starting at the first block which has a non-empty carry space as determined by its degree of fullness.

3.5.6 Complexity of Carry Propagation

We now give the complexities of the various carry propagation algorithms.

Algorithm 37: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathtt{FullPropagate}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}\right)$

Input: $\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$: a homomorphic integer with potentially non-empty carry spaces
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$: a homomorphic integer with empty carry spaces
/* Get the index of the first block with degree above $\mathsf{msg}-1$ */
$\mathsf{start} \leftarrow \mathsf{FirstBlockWithDegreeAbove}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}, \mathsf{msg}-1\right)$
for $i \in [0, \mathsf{start}-1]$ **do**
  $\mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{ct}_i^{(\mathsf{in})}$
$\mathsf{ct}_{\mathsf{start}}^{(\mathsf{out})} \leftarrow \mathsf{PBS}\left((x) \mapsto x \bmod \mathsf{msg}, \mathsf{ct}_{\mathsf{start}}^{(\mathsf{in})}\right)$
/* This can be done in parallel */
for $i \in [0, \mathcal{B}-\mathsf{start}-2]$ **do**
  $\mathsf{ct}_i^{(\mathsf{messages})} \leftarrow \mathsf{PBS}\left((x) \mapsto x \bmod \mathsf{msg}, \mathsf{ct}_{i+\mathsf{start}+1}^{(\mathsf{in})}\right)$
  $\mathsf{ct}_i^{(\mathsf{carries})} \leftarrow \mathsf{PBS}\left((x) \mapsto \left\lfloor \frac{x}{\mathsf{msg}} \right\rfloor, \mathsf{ct}_{i+\mathsf{start}}^{(\mathsf{in})}\right)$
$\overrightarrow{\mathsf{ct}}^{(\mathsf{tmp})} \leftarrow \overrightarrow{\mathsf{ct}}^{(\mathsf{messages})} + \overrightarrow{\mathsf{ct}}^{(\mathsf{carries})}$
/* This will use $\mathcal{B}-\mathsf{start}-1$ in place of $\mathcal{B}$ */
$\overrightarrow{\mathsf{ct}}^{(\mathsf{propagated})} \leftarrow \mathsf{PropagateSingleBitCarries}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{tmp})}\right)$
/* This is just relabelling */
for $i \in [\mathsf{start}+1, \mathcal{B}-1]$ **do**
  $\mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{ct}_{i-\mathsf{start}-1}^{(\mathsf{propagated})}$
return $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

Theorem 16 *Let $\mathsf{ct}^{(\mathsf{in})}$ be a large homomorphic integer. Applying the sequential carry propagation of Algorithm 28 to $\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$ has complexity $\mathbb{C}^{\mathsf{SequentialCarryPropagation}} = \mathbb{C}^{\mathsf{PBS}} \cdot 2 \cdot \mathcal{B}$.*

Theorem 17 *Let $\mathsf{ct}^{(\mathsf{in})}$ be a large homomorphic integer where each block encrypts a value that is at most $2 \cdot \mathsf{msg} - 2$. Let $G = \log_2(\mathsf{p})$ be the group size and $L(\mathcal{B}) = \left\lceil \frac{\mathcal{B}}{G} \right\rceil$ be the function returning the number of groups given a number of blocks as input.*

When using sequential inner propagation, the complexity of Algorithm 36:

$$\mathbb{C}^{\mathsf{PropagateSingleBitCarries}}(\mathcal{B}) = \mathbb{C}^{\underline{\mathtt{BivPBS}}} \cdot (\mathcal{B} + \mathcal{B} - 1 + \max(0, L(\mathcal{B}) - 2) + \mathcal{B})$$

When using Hillis-Steele inner propagation, the complexity is:

$$\mathbb{C}^{\mathsf{PropagateSingleBitCarries}}(\mathcal{B}) = \mathbb{C}^{\underline{\mathtt{BivPBS}}} \cdot (\mathcal{B} + \mathcal{B} - 1 + \mathcal{B}) + \mathbb{C}^{\mathsf{Hillis-Steele}}(L(\mathcal{B}) - 1)$$

With $\mathbb{C}^{\mathsf{Hillis-Steele}}(x) = \mathbb{C}^{\underline{\mathtt{BivPBS}}} \cdot \sum_{d=0}^{\lceil \log_2(x) \rceil} \left(x - 2^d\right) = \mathbb{C}^{\underline{\mathtt{BivPBS}}} \cdot \left((\lceil \log_2(x) \rceil + 1) \cdot x + 1 - 2^{1+\lceil \log_2(x) \rceil} \right)$.

Theorem 18 *Let $\mathsf{ct}^{(\mathsf{in})}$ be a large homomorphic integer having $\mathcal{B}$ blocks, assuming that we already have used the carry space in the first block and we want to empty all the carries using Algorithm 37 then the complexity of this operation is:*

$$\mathbb{C}^{\mathsf{FullPropagate}}(\mathcal{B}) = \mathbb{C}^{\underline{\mathsf{PBS}}} \cdot (2 \cdot \mathcal{B} - 1) + \mathbb{C}^{\mathsf{PropagateSingleBitCarries}}(\mathcal{B} - 1).$$

Since Algorithm 37 benefits greatly from parallelism, it is also instructive to see how it compares with Algorithm 28 in terms of the sequential depth of PBSes performed. In this regard, the sequential algorithm has depth $\mathcal{B}$ PBSes while the parallel algorithm has depth either

$$4 + \lceil \max(\lceil \mathcal{B} / \log_2(\mathsf{p}) \rceil - 2, 0) / (\log_2(\mathsf{p}) - 1) \rceil$$

if using the sequential algorithm to resolve group carries (Algorithm 33), or

$$4 + \lceil \log_2(\max((\lceil \mathcal{B} / \log_2(\mathsf{p}) \rceil - 1), 1)) \rceil$$

if using Hillis-Steele (Algorithm 35).

3.6 Addition

In TFHE, ciphertexts and plaintexts can be freely added as long as certain constraints are met – namely, noise constraints and constraints on the degrees of fullness (see Section 3.2.1). This addition naturally extends to homomorphic integers. However, in the radix representation, addition becomes a costly operation due to the need for carry propagation, as explained in Section 3.5.

3.6.1 Ciphertext-Ciphertext Addition

The addition of homomorphic integers is presented in Algorithm 38, and Theorem 19 states its guarantees and complexity. Note that the computations at each step of the loop can be performed in parallel across all blocks.

Algorithm 38: $\overrightarrow{\mathsf{ct}}^{(\mathsf{Add})} \leftarrow \mathsf{Add}\left(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\right)$

Input: $\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}$: two homomorphic integers
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{Add})}$

1 **for** $i \in [0, \mathcal{B} - 1]$ **do**
2 $\quad \mathsf{ct}_i^{(\mathsf{Add})} \leftarrow \mathsf{ct}_i^{(1)} + \mathsf{ct}_i^{(2)}$
3 **return** $\mathtt{FullPropagate}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{Add})}\right)$

Theorem 19 *Let $\overrightarrow{\mathsf{ct}}^{(1)}$ and $\overrightarrow{\mathsf{ct}}^{(2)}$ be two large homomorphic integers. Then,*

$$\mathsf{Dec}_{\mathsf{s}}\left(\mathsf{Add}\left(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\right)\right) = m^{(1)} + m^{(2)} \pmod{\Omega}$$

The algorithm's complexity is: $\mathbb{C}^{\mathsf{Add}} = \mathbb{C}^{\mathtt{FullPropagate}}$.

3.6.2 Plaintext-Ciphertext Addition

To add a plaintext to a homomorphic integer, we can directly use Algorithm 38 by interpreting the plaintext as a trivial ciphertext (see Definition 6). In TFHE-rs, a more efficient version of this algorithm is implemented when one operand is a trivial ciphertext and will be included in future versions of this document.

3.7 Negation

Similarly to the case of addition, the negation operation on LWE ciphertexts extends naturally to homomorphic integers. The negation itself is computationally inexpensive compared to the costly final carry propagation required to reset the encoding. The full algorithm is presented in Algorithm 39 and Theorem 20 states its guarantees and complexity.

Algorithm 39: $\overrightarrow{\mathsf{ct}}^{(\mathsf{Neg})} \leftarrow \mathsf{Neg}\left(\overrightarrow{\mathsf{ct}}\right)$

Input: $\overrightarrow{\mathsf{ct}}$: a large homomorphic integer
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{Neg})}$: a large homomorphic integer

$z_b \leftarrow 0$
for $i \in [0, \mathcal{B}-1]$ **do**
  $\mathsf{block} \leftarrow \mathsf{ct}_i + z_b$
  $z \leftarrow \max\left\{\left\lceil \frac{\deg(\mathsf{block})}{\mathsf{msg}} \right\rceil, 1\right\} \cdot \mathsf{msg}$
  $\mathsf{ct}_i^{(\mathsf{Neg})} \leftarrow (-\mathsf{block}) + z$
  $z_b \leftarrow z \,//\, \mathsf{msg}$
return `FullPropagate`$\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{Neg})}\right)$

Theorem 20 *Let $\overrightarrow{\mathsf{ct}}$ be a large homomorphic integer encrypting $m \in \mathbb{Z}_\Omega$. Then,*

$$\mathsf{Dec}_\mathsf{s}\left(\mathsf{Neg}\left(\overrightarrow{\mathsf{ct}}\right)\right) = -m \pmod{\Omega}.$$

The algorithm's complexity is $\mathbb{C}^{\mathsf{Neg}} = \mathbb{C}^{\mathtt{FullPropagate}}$.

Remark 34 *Similar to the addition algorithm, the negation algorithm can take as input a homomorphic integer with non-empty carry buffers. However, a certain amount of carry space must still be available. The required carry space for negation depends on the degrees of fullness of the preceding blocks (notably, some blocks may be completely full, i.e., $\deg(\mathsf{ct_i}) = \mathsf{carry} \cdot \mathsf{msg} - 1$). As a result, it is not possible to determine a tight bound on the degrees of fullness.*

3.8 Subtraction

The subtraction presented in Algorithm 40 comes naturally from the homomorphic addition of LWE ciphertexts. Subtraction can be implemented in two ways:

- Adding the negation, i.e., $a - b = a + (-b)$, only then the carries are propagated (once),
- Direct subtraction with a correcting term, then borrows are propagated.

In TFHE-rs, subtraction is performed by adding one operand to the negation of the second, except for overflow detection in unsigned subtraction, where the second method is used instead (see Section 3.14.2).

Algorithm 40: $\overrightarrow{\mathsf{ct}}^{(\mathsf{Sub})} \leftarrow \boxed{\mathsf{Sub}\left(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\right)}$ $\boxed{\mathsf{Sub}\left(\overrightarrow{\mathsf{ct}}^{(1)}, v\right)}$

Input: $\left\{\boxed{\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\text{: two homomorphic integers}}\right.$ $\boxed{\overrightarrow{\mathsf{ct}}^{(1)}\text{, v: a homomorphic integer and a scalar}}$
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{Sub})}$: large homomorphic integer

$\boxed{\overrightarrow{\mathsf{ct}}^{(\mathsf{Neg})} \leftarrow \mathsf{Neg}\left(\overrightarrow{\mathsf{ct}}^{(2)}\right)}$; // without the final `FullPropagate()`
$\boxed{\overrightarrow{\mathsf{ct}}^{(\mathsf{Sub})} \leftarrow \mathsf{Add}\left(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(\mathsf{Neg})}\right)}$ $\boxed{\overrightarrow{\mathsf{ct}}^{(\mathsf{Sub})} \leftarrow \mathsf{Add}\left(\overrightarrow{\mathsf{ct}}^{(1)}, -v\right)}$
return $\overrightarrow{\mathsf{ct}}^{(\mathsf{Sub})}$

Theorem 21 states the guarantees of Algorithm 40 and its complexity.

Theorem 21 *Let* $\overrightarrow{\mathsf{ct}}^{(1)}$ *and* $\overrightarrow{\mathsf{ct}}^{(2)}$ *be two large homomorphic integers that encrypt* $m^{(1)}$ *and* $m^{(2)}$*, respectively. Then,*

$$\mathsf{Dec}_{\mathsf{s}}\Big(\mathsf{Sub}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big)\Big) = m^{(1)} - m^{(2)} \pmod{\Omega}$$

The algorithm's complexity is $\mathbb{C}^{\mathsf{Sub}} = \mathbb{C}^{\mathsf{Add}}$.

3.8.1 Plaintext-Ciphertext Subtraction

To subtract a plaintext from a homomorphic integer, we can negate the scalar and use the plaintext-ciphertext addition algorithm discussed in Section 3.6.2.

3.9 Block Shifts and Rotations

Shifts and rotations can be done at two levels, *blocks* or *bits*. Operating at the block level is often more useful as a building block for other operations rather than for end users which will use the bit versions as the blocks are abstracted. In this section, we describe plaintext-ciphertext and ciphertext-ciphertext *block* shifts while bit shifts are described in Section 3.10. The operation descriptions in this section begin with Plaintext-Ciphertext block shifts, as some of their subroutines are required to define the Ciphertext-Ciphertext case.

3.9.1 Plaintext-Ciphertext Block Shift & Rotation

Shifting and rotating blocks by a known amount are very cheap operations as everything happens in the clear, and it involves no FHE operations. We present four algorithms for both left and right shift and rotate in Algorithm 41.

Algorithm 41: Block Shift / Rotate By A Clear Amount

Input: $\begin{cases} \overrightarrow{\mathsf{ct}}^{(\mathsf{in})} : \text{a homomorphic integer} \\ \mathsf{blockShift} \in [0, \mathcal{B}-1] : \text{the number of blocks to shift} \end{cases}$

Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

Function `BlocksShiftLeft`($\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$, blockShift):
- **for** $i \in [0, \mathcal{B} - \mathsf{blockShift} - 1]$ **do**
  - $j \leftarrow i + \mathsf{blockShift}$
  - $\mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{ct}_j^{(\mathsf{in})}$
- **for** $i \in [\mathcal{B} - \mathsf{blockShift}, \mathcal{B} - 1]$ **do**
  - $\mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{ct}^{(\mathsf{Triv})}(0)$
- **return** $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

Function `BlocksShiftRight`($\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$, blockShift):
- **for** $i \in [0, \mathsf{blockShift} - 1]$ **do**
  - $\mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{ct}^{(\mathsf{Triv})}(0)$
- **for** $i \in [\mathsf{blockShift}, \mathcal{B} - 1]$ **do**
  - $j \leftarrow i - \mathsf{blockShift}$
  - $\mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{ct}_j^{(\mathsf{in})}$
- **return** $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

Function `BlocksRotateLeft`($\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$, blockShift):
- **for** $i \in [0, \mathcal{B} - 1]$ **do**
  - $\mathsf{j} \leftarrow (\mathsf{blockShift} + i) \bmod \mathcal{B}$
  - $\mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{ct}_j^{(\mathsf{in})}$
- **return** $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

Function `BlocksRotateRight`($\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$, blockShift):
- **for** $i \in [0, \mathcal{B} - 1]$ **do**
  - $\mathsf{j} \leftarrow (\mathcal{B} - \mathsf{blockShift} + i) \bmod \mathcal{B}$
  - $\mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{ct}_j^{(\mathsf{in})}$
- **return** $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

3.9.2 Ciphertext-Ciphertext Block Shift & Rotation

Shifting or rotating blocks by an encrypted amount introduces additional complexity, as it requires FHE operations. The algorithm is based on the *Barrel shifter* electronic circuit.

First, the bits of the integer encrypting the amount to shift or rotate by are extracted into individual blocks using a PBS. Only the necessary bits are extracted, that is, only the bits in a position that is in the range $[0, \lceil \log_2(\mathcal{B}) \rceil]$, and these encrypted bits make up $\overrightarrow{\mathsf{ct}}^{(\mathsf{shiftBits})}$.

Then, for each block in the ciphertext to be shifted/rotated, two bivariate PBSes are performed, one to extract its new *message* value and another to extract the value that is meant to go to the next block (called *carry* here). Then, depending on the current iteration, the *carry* is aligned with the *message*, and the two are added together to form the new block value.

The algorithm for the set-up of the look-up table is outlined in Algorithm 42. The full algorithm for encrypted block shift and rotation is outlined in Algorithm 43.

$$\mathsf{shiftMessage}(\mathsf{x}, \mathsf{shiftBit}) = \begin{cases} x, & \text{if } \mathsf{shiftBit} = 0 \\ 0, & \text{otherwise} \end{cases} \qquad \mathsf{shiftCarry}(\mathsf{x}, \mathsf{shiftBit}) = \begin{cases} 0, & \text{if } \mathsf{shiftBit} = 0 \\ x, & \text{otherwise} \end{cases} \tag{34}$$

Algorithm 42: Auxiliary function for creation of LUT for the block shift

Input: $\begin{cases} \mathsf{lastBlock} \in [0, \mathsf{msg} - 1] \\ \mathsf{shiftBit} \in \{0, 1\} \end{cases}$
Output: $\mathsf{result} \in \{0, 1\}$

```
Function PaddingBlockLut(lastBlock, shiftBit):
    if shiftBit = 1 then
        signBit ← lastBlock // (msg // 2)
        result = (msg − 1) · signBit
    else
        result = 0
    return result
```

Algorithm 43: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathsf{BlockBarrelShift}\Big(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{shiftBits})}, \mathsf{kind}, \mathsf{range}\Big)$

Input: $\begin{cases} \overrightarrow{\mathsf{ct}}^{(\mathsf{in})} : \text{ a homomorphic integer} \\ \overrightarrow{\mathsf{ct}}^{(\mathsf{shiftBits})} : \text{ each block encrypts 1 bit, i.e., } \overrightarrow{\mathsf{ct}}^{(\mathsf{shiftBits})} \leftarrow \texttt{ExtractBits}() \\ \mathsf{kind} \in \{\mathsf{LeftShift}, \mathsf{LeftRotate}, \mathsf{RightShift}, \mathsf{RightRotate}\} : \text{ a type for the shift} \\ \mathsf{range} \in [0, \mathcal{B}-1] : \text{the size of the shift} \end{cases}$

Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

$j \leftarrow 0$
for $d \in \mathsf{range}$ **do**
  if kind = RightShift AND isSigned($\mathsf{ct}^{(\mathsf{in})}$) **then**
    $\mathsf{paddingBlock} \leftarrow \mathsf{BivPBS}\Big(\texttt{PaddingBlockLut}(\cdot,\cdot), \mathsf{ct}^{(\mathsf{in})}_{\mathcal{B}-1}, \mathsf{ct}^{(\mathsf{shiftBits})}_j\Big)$
  else
    $\mathsf{paddingBlock} \leftarrow \mathsf{ct}^{(\mathsf{Triv})}(0))$
  $\mathsf{carries} \leftarrow \{\}$
  for $i \in [0, \mathcal{B}-1]$ **do**
    /* $\mathsf{ct}^{(\mathsf{shiftBits})}$ is already shifted by msg */
    $\mathsf{ct}^{(\mathsf{out})}_i \leftarrow \mathsf{PBS}\Big(\mathsf{shiftMessage}(\cdot,\cdot),\ \mathsf{ct}^{(\mathsf{in})}_i + \mathsf{ct}^{(\mathsf{shiftBits})}_j\Big)$ // defined in Eq. (34)
    $\mathsf{carries.append}\Big(\mathsf{PBS}\Big(\mathsf{shiftCarry}(\cdot,\cdot),\ \mathsf{ct}^{(\mathsf{in})}_i + \mathsf{ct}^{(\mathsf{shiftBits})}_j\Big)\Big)$ // defined in Eq. (34)
  $j \leftarrow j + 1;$
  switch kind **do**
    case LeftShift **do**
      $\mathsf{carries} \leftarrow \texttt{BlocksRotateRight}(\mathsf{carries}, (1 \ll d) - 1)$
      for $i \in [0, 1 \ll d]$ **do**
        $\mathsf{carries}[i] \leftarrow \mathsf{paddingBlock}$
    case RightShift **do**
      $\mathsf{carries} \leftarrow \texttt{BlocksRotateLeft}(\mathsf{carries}, 1 \ll d)$
      for $i \in [\mathcal{B} - (1 \ll d), \mathcal{B}-1]$ **do**
        $\mathsf{carries}[i] \leftarrow \mathsf{paddingBlock}$
    case LeftRotate **do**
      $\mathsf{carries} \leftarrow \texttt{BlocksRotateRight}(\mathsf{carries}, 1 \ll d)$
    case RightRotate **do**
      $\mathsf{carries} \leftarrow \texttt{BlocksRotateLeft}(\mathsf{carries}, 1 \ll d)$
    for $i \in [0, \mathcal{B}-1]$ **do**
      $\mathsf{ct}^{(\mathsf{out})}_i \leftarrow \mathsf{ct}^{(\mathsf{out})}_i + \mathsf{carries}[i]$
/* Clean the noise */
for $i \in [0, \mathcal{B}-1]$ **do**
  $\mathsf{ct}^{(\mathsf{out})}_i \leftarrow \mathsf{PBS}\Big((x) \mapsto x \bmod \mathsf{msg}, \mathsf{ct}^{(\mathsf{out})}_i\Big)$
return $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

The algorithms are defined as a specific call of the `BlockBarrelShift` algorithm, i.e.:

$$\mathsf{BlockLeftShift}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big) = \texttt{BlockBarrelShift}(\overrightarrow{\mathsf{ct}}^{(1)}, \texttt{ExtractBits}(\overrightarrow{\mathsf{ct}}^{(2)}, \log_2(\mathsf{msg}), \lceil\log_2(\mathcal{B})\rceil), \mathsf{LeftShift}, [0, \lceil\log_2(\mathcal{B})\rceil])$$

$$\mathsf{BlockRightShift}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big) = \texttt{BlockBarrelShift}(\overrightarrow{\mathsf{ct}}^{(1)}, \texttt{ExtractBits}(\overrightarrow{\mathsf{ct}}^{(2)}, \log_2(\mathsf{msg}), \lceil\log_2(\mathcal{B})\rceil), \mathsf{RightShift}, [0, \lceil\log_2(\mathcal{B})\rceil])$$

$$\mathsf{BlockLeftRotate}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big) = \texttt{BlockBarrelShift}(\overrightarrow{\mathsf{ct}}^{(1)}, \texttt{ExtractBits}(\overrightarrow{\mathsf{ct}}^{(2)}, \log_2(\mathsf{msg}), \lceil\log_2(\mathcal{B})\rceil), \mathsf{LeftRotate}, [0, \lceil\log_2(\mathcal{B})\rceil])$$

$$\mathsf{BlockRightRotate}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big) = \texttt{BlockBarrelShift}(\overrightarrow{\mathsf{ct}}^{(1)}, \texttt{ExtractBits}(\overrightarrow{\mathsf{ct}}^{(2)}, \log_2(\mathsf{msg}), \lceil\log_2(\mathcal{B})\rceil), \mathsf{RightRotate}, [0, \lceil\log_2(\mathcal{B})\rceil])$$

Theorem 22 *Let* $\overrightarrow{\mathsf{ct}}^{(1)}$ *and* $\overrightarrow{\mathsf{ct}}^{(2)}$ *be two large homomorphic integers.*

Then,

$$\begin{aligned}
\mathsf{Dec_s}\Big(\mathsf{BlockLeftShift}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big)\Big) &= (m^{(1)} \cdot \mathsf{msg}^{m^{(2)}}) \pmod{\Omega},\\
\mathsf{Dec_s}\Big(\mathsf{BlockRightShift}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big)\Big) &= (m^{(1)} \mathbin{//} \mathsf{msg}^{m^{(2)}}) \pmod{\Omega},\\
\mathsf{Dec_s}\Big(\mathsf{BlockLeftRotate}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big)\Big) &= (m^{(1)} \cdot \mathsf{msg}^{m^{(2)}}) \mid (m^{(1)} \mathbin{//} \mathsf{msg}^{\mathcal{B}-m^{(2)}}) \pmod{\Omega},\\
\mathsf{Dec_s}\Big(\mathsf{BlockRightRotate}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big)\Big) &= (m^{(1)} \mathbin{//} \mathsf{msg}^{m^{(2)}}) \mid (m^{(1)} \cdot \mathsf{msg}^{\mathcal{B}-m^{(2)}}) \pmod{\Omega}.
\end{aligned}$$

Let $\eta = \lceil \log_2(\mathcal{B}) \rceil$ *be the number of bits that represent the number of block shifts.*

$$\mathbb{C} = \mathbb{C}^{\mathsf{PBS}} \cdot \left(\left\lceil \frac{\eta}{\log_2(\mathsf{msg})} \right\rceil + (2 \cdot \eta \cdot \mathcal{B}) + \mathcal{B} \right)$$

When using the ManyLUT PBS *(see Algorithm 21), which is possible only for* $\mathsf{msg} > 2$*, the complexity of unsigned left/right shifts and rotations, as well as signed left/right rotations and signed left shifts, is given by:*

$$\mathbb{C} = \mathbb{C}^{\mathsf{PBS}} \cdot \left(\left\lceil \frac{\eta}{\log_2(\mathsf{msg})} \right\rceil + (\eta \cdot \mathcal{B}) + \mathcal{B} \right),$$

The complexity of the signed right shift is given by:

$$\mathbb{C} = \mathbb{C}^{\mathsf{PBS}} \cdot \left(\left\lceil \frac{\eta}{\log_2(\mathsf{msg})} \right\rceil + (\eta \cdot (\mathcal{B}+1)) + \mathcal{B} \right).$$

3.10 Bit Shifts and Rotations

In the previous section, we explored how to perform shifts and rotations on the blocks that compose a homomorphic integer. In this section, we introduce a method for shifting and rotating directly on the bits of the large integer, abstracting away the underlying block layout.

3.10.1 Plaintext-Ciphertext Bit Shift & Rotation

Shifting or rotating bits by a known amount is a rather easy operation that only requires one layer of bivariate PBS that can be done in parallel.

The scalar can be decomposed with $\upsilon = \log_2(\mathsf{msg}) \cdot k + r$ where k gives by how many places the blocks should be shifted/rotated and r gives by how many places the bits shall move between blocks. By definition, we have $r \in [0, \log_2(\mathsf{msg}) - 1]$, meaning that bits can only shift between adjacent blocks. Thus, using a bivariate PBS between a block and its predecessor or successor (depending on the shift direction) allows us to encode this behavior. As for shifting/rotating blocks, this operation is done in the clear and does not require PBS: it is cheap and fast.

The plaintext-ciphertext left shift is detailed in Algorithm 45, and the plaintext-ciphertext left rotation in Algorithm 46, both of which rely on auxiliary functions defined in Algorithm 44. Similarly, the plaintext-ciphertext right shift is detailed in Algorithm 48, and the plaintext-ciphertext right rotation in Algorithm 49, both relying on auxiliary functions given in Algorithm 47.

Remark 35 (Signed Integers) *For signed integers, the right shift is slightly different: the* arithmetic *shift is performed. In the arithmetic shift, instead of filling the most significant bits with* 0 *(as in the* logical *shift), the most significant bits are filled with the value of the* sign bit*, requiring to slightly adapt the function used in the different* PBS*es.*

Theorem 23 *Let $\overrightarrow{\mathsf{ct}}^{(1)}$ be a large homomorphic integer and let $\upsilon = \log_2(\mathsf{msg}) \cdot k + r$. Then,*

$$\mathsf{Dec_s}\Big(\mathsf{LeftShift}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \upsilon\Big)\Big) = m^{(1)} \ll \upsilon = (m^{(1)} \cdot 2^{\upsilon}) \pmod{\Omega},$$
$$\mathsf{Dec_s}\Big(\mathsf{RightShift}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \upsilon\Big)\Big) = m^{(1)} \gg \upsilon = (m^{(1)} \mathbin{//} 2^{\upsilon}) \pmod{\Omega},$$
$$\mathsf{Dec_s}\Big(\mathsf{RightRotate}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \upsilon\Big)\Big) = (m^{(1)} \ll \upsilon) \mid (m^{(1)}(\gg \log_2(\Omega) - \upsilon)),$$
$$\mathsf{Dec_s}\Big(\mathsf{LeftRotate}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \upsilon\Big)\Big) = (m^{(1)} \gg \upsilon) \mid (m^{(1)}(\ll \log_2(\Omega) - \upsilon)).$$

Setting

$$\delta(x) = \begin{cases} 0 & \text{if } x = 0, \\ 1 & \text{otherwise}, \end{cases}$$

the complexity of plaintext-ciphertext left or right shift is:

$$\mathbb{C}^{Shift} = (\mathcal{B} - k) \cdot \delta(r) \cdot \mathbb{C}^{\mathsf{PBS}},$$

and the complexity of plaintext-ciphertext left or right is:

$$\mathbb{C}^{Rot} = \mathcal{B} \cdot \delta(r) \cdot \mathbb{C}^{\mathsf{PBS}}$$

.

Algorithm 44: Auxiliary functions for plaintext-ciphertext left/right shift and rotate

Input: $\begin{cases} \mathsf{previous}, \mathsf{current}, \mathsf{next} \in [0, \mathsf{msg} - 1] : \text{ input domains to define the } \lambda\text{-functions} \\ \mathsf{shiftBetweenBlocks} \in [0, \log_2(\mathsf{msg}) - 1] : \text{ a known constant value} \end{cases}$
Output: result $\in \{0, 1\}$

```
Function LeftShiftBetweenBlocks(previous, current, shiftBetweenBlocks):
    bitsFromPrevious ← previous ≫ shiftBetweenBlocks
    bitsFromCurrent ← (current ≪ shiftBetweenBlocks) mod msg
    result = bitsFromCurrent OR bitsFromPrevious
    return result
Function RightShiftBetweenBlocks(current, next, shiftBetweenBlocks):
    bitsFromCurrent ← current ≫ shiftBetweenBlocks
    bitsFromNext ← (next ≪ shiftBetweenBlocks) mod msg
    result = bitsFromCurrent OR bitsFromPrevious
    return result
```

3.10.2 Ciphertext-Ciphertext Bit Shift & Rotation

For the shifts and rotations by an encrypted amount, the computations are more complex, and there are a few different possibilities depending on msg:

- If msg = 2, then each block encrypts a single bit, thus bit shift/rotation is the same as block shift/rotation.
- If $\log_2(\mathsf{msg})$ is a power of 2, an algorithm based on the decomposition described in Section 3.10.1 can be used.
- Otherwise, a barrel shifter can be applied.

Algorithm 45: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathsf{LeftShift}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}, v\right)$

Input: $\begin{cases} \overrightarrow{\mathsf{ct}}^{(\mathsf{in})} \text{: homomorphic integer} \\ v \in [0, \mathcal{B} \cdot \log_2(\mathsf{msg}) - 1] \end{cases}$
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$: homomorphic integer

$\mathsf{blockShift} \leftarrow v \mathbin{//} \log_2(\mathsf{msg})$
$\mathsf{shiftBetweenBlocks} \leftarrow v \bmod \log_2(\mathsf{msg})$
if $\mathsf{shiftBetweenBlocks} = 0$ **then**
  $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathtt{BlocksShiftRight}(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}, \mathsf{blockShift})$
else
  $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \left\{\mathsf{ct}^{(\mathsf{Triv})}(0), \cdots, \mathsf{ct}^{(\mathsf{Triv})}(0)\right\}^{\mathsf{blockShift}}$
  $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}.\mathsf{append}\left(\mathsf{PBS}\left((x) \mapsto (x \ll \mathsf{shiftBetweenBlocks}) \bmod \mathsf{msg}, \overrightarrow{\mathsf{ct}}_0^{(\mathsf{in})}\right)\right)$
  for $i \in [1, \mathcal{B} - \mathsf{blockShift} - 1]$ **do**
    $\mathsf{lut} \leftarrow \mathbf{GenLUT}\left((\mathsf{prev}, \mathsf{curr}) \mapsto \mathtt{LeftShiftBetweenBlocks}(\mathsf{prev}, \mathsf{curr}, \mathsf{shiftBetweenBlocks})\right)$
    $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}.\mathsf{append}\left(\mathsf{BivPBS}\left(\mathsf{lut}, \mathsf{ct}_{i-1}^{(\mathsf{in})}, \mathsf{ct}_i^{(\mathsf{in})}\right)\right)$
return $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

Recall that we have assumed that $\log_2(\mathsf{msg})$ is a power-of-2 so the third case is not applicable here.

Remark 36 (Signed Integers.) *As with the plaintext-ciphertext shift (Section 3.10.1), the right shift will perform an arithmetic shift, thus the functions used in that case need to be modified slightly.*

Given an encryption $\overrightarrow{\mathsf{ct}}^{(\mathsf{amount})}$ of $m^{(\mathsf{amount})}$, the amount we want to shift/rotate by, we can homomorphically decompose it into encryptions of

$$\mathsf{numBitShift} = m^{(\mathsf{amount})} \bmod \log_2(\mathsf{msg}), \text{ and of}$$
$$\mathsf{numBlockShift} = m^{(\mathsf{amount})} \mathbin{//} \log_2(\mathsf{msg}).$$

As seen above, these computations involve homomorphic division and modulo operations, both of which have not yet been described. By analogy with hardware and big-integer libraries, they can be assumed to be among the most time-consuming operations. However, since we know that $\log_2(\mathsf{msg})$ is a power of two, the integer division // is equivalent to a bit shift, and the modulo operation mod corresponds to a *bitwise AND* (&).

Thus, to perform ciphertext-ciphertext shifting or rotation, we first compute numBitShift and numBlockShift as explained above. Then, we shift the bits by numBitShift, followed by shifting the blocks by numBlockShift. Since $\mathsf{numBitShift} \in [0, \log_2(\mathsf{msg}) - 1]$ and each block encrypts a value in $[0, \mathsf{msg} - 1]$, the bivariate PBS can be used to shift bits between blocks.

Unlike the plaintext-ciphertext version of this algorithm, performing the shift requires two PBS operations per block, along with an addition.

Let amountBlock be the first block of $\overrightarrow{\mathsf{ct}}^{(\mathsf{amount})}$. Informally, to perform a ciphertext-ciphertext left shift, we first compute, for each block in $\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$, the new messages after applying numBitShift using the function:

$$(x, \mathsf{amountBlock}) \mapsto (x \ll (\mathsf{amountBlock} \bmod \log_2(\mathsf{msg}))) \bmod \mathsf{msg}.$$

Next, we compute the carries using the function:

$$(x, \mathsf{amountBlock}) \mapsto x \gg (\log_2(\mathsf{msg}) - (\mathsf{amountBlock} \bmod \log_2(\mathsf{msg}))).$$

Algorithm 46: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathsf{LeftRotate}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}, \upsilon\right)$

Input: $\begin{cases} \overrightarrow{\mathsf{ct}}^{(\mathsf{in})} : \text{homomorphic integer} \\ \upsilon \in [0, \mathcal{B} \cdot \log_2(\mathsf{msg}) - 1] \end{cases}$
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$: homomorphic integer

$\mathsf{blockShift} \leftarrow \upsilon \,//\, \log_2(\mathsf{msg})$
$\mathsf{shiftBetweenBlocks} \leftarrow \upsilon \bmod \log_2(\mathsf{msg})$
$\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \texttt{BlocksRotateRight}(\overrightarrow{\mathsf{ct}}^{(\mathsf{out})})$
$\mathsf{blockShift} \leftarrow \upsilon \,//\, \mathsf{msg}$
$\mathsf{shiftBetweenBlocks} \leftarrow \upsilon \bmod \mathsf{msg}$
$\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \texttt{BlocksRotateRight}(\overrightarrow{\mathsf{ct}}^{(\mathsf{out})})$
if $\mathsf{shiftBetweenBlocks} \neq 0$ **then**
  $\mathsf{lut} \leftarrow \mathbf{GenLUT}\Big((\mathsf{prev}, \mathsf{curr}) \mapsto \texttt{LeftShiftBetweenBlocks}(\mathsf{prev}, \mathsf{curr}, \mathsf{shiftBetweenBlocks})\Big)$
  for $i \in [0, \mathcal{B} - 1]$ **do**
    $\mathsf{prev} \leftarrow$ **if** $\mathsf{i} = 0$ **then** $\mathcal{B} - 1$ **else** $\mathsf{i} - 1$
    $\mathsf{ct}_i^{(\mathsf{out})} \leftarrow \mathsf{BivPBS}\Big(\mathsf{lut}, \mathsf{ct}_{\mathsf{prev}}^{(\mathsf{out})}, \mathsf{ct}_{\mathsf{i}}^{(\mathsf{out})}\Big)$
return $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

Algorithm 47: Auxiliary functions plaintext-ciphertext right rotate and shift

Input: $\mathsf{mostSignificantBlock} \in [0, \mathsf{msg} - 1]$: input domain to define the λ-functions
Output: $\mathsf{result} \in [0, \mathsf{msg} - 1]$
Function `RightShiftSignedBlock(mostSignificantBlock):`
  $\mathsf{signBit} \leftarrow \mathsf{mostSignificantBlock} \,//\, (\mathsf{msg} \,//\, 2)$
  $\mathsf{padding} \leftarrow (\mathsf{msg} - 1) \cdot \mathsf{signBit}$
  $\mathsf{fullBlock} \leftarrow (\mathsf{padding} \cdot \mathsf{msg}) \mid \mathsf{mostSignificantBlock}$
  $\mathsf{result} = (\mathsf{fullBlock} \ll \mathsf{shiftBetweenBlocks}) \bmod \mathsf{msg}$
  return result

Input: $\mathsf{mostSignificantBlock} \in [0, \mathsf{msg} - 1]$: input domain to define the λ-functions
Output: $\mathsf{padding} \in [0, \mathsf{msg} - 1]$
Function `RightShiftSignedPaddingBlocks(mostSignificantBlock):`
  $\mathsf{signBit} \leftarrow \mathsf{mostSignificantBlock} \,//\, (\mathsf{msg} \,//\, 2)$
  $\mathsf{padding} \leftarrow (\mathsf{msg} - 1) \cdot \mathsf{signBit}$
  return padding

Algorithm 48: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathsf{RightShift}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}, \upsilon\right)$

Input: $\begin{cases} \overrightarrow{\mathsf{ct}}^{(\mathsf{in})} : \text{homomorphic integer} \\ \upsilon \in [0, \mathcal{B} \cdot \log_2(\mathsf{msg}) - 1] \end{cases}$
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} : \text{homomorphic integer}$

$\mathsf{blockShift} \leftarrow \upsilon \,//\, \log_2(\mathsf{msg})$
$\mathsf{shiftBetweenBlocks} \leftarrow \upsilon \bmod \log_2(\mathsf{msg})$
if $\mathsf{shiftBetweenBlocks} = 0$ **then**
  $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathtt{BlocksShiftLeft}(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})})$
else
  $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \{\}$
  $\mathsf{lut} \leftarrow \mathbf{GenLUT}\Big((\mathsf{current}, \mathsf{next}) \mapsto \mathtt{RightShiftBetweenBlocks}(\mathsf{current}, \mathsf{next}, \mathsf{shiftBetweenBlocks})\Big)$
  for $i \in [\mathsf{blockShift}, \mathcal{B} - 2]$ **do**
    $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}.\mathsf{append}\Big(\mathsf{BivPBS}\Big(\mathsf{lut}, \overrightarrow{\mathsf{ct}}_i^{(\mathsf{in})}, \overrightarrow{\mathsf{ct}}_{i+1}^{(\mathsf{in})}\Big)\Big)$
  if $\mathsf{isSigned}(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})})$ **then**
    $\mathsf{lut} \leftarrow \mathbf{GenLUT}\big(\mathtt{RightShiftSignedBlock}(\cdot)\big)$
    $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}.\mathsf{append}\Big(\mathsf{PBS}\Big(\mathsf{lut}, \overrightarrow{\mathsf{ct}}_{\mathcal{B}-1}^{(\mathsf{in})}\Big)\Big)$
    $\mathsf{lut} \leftarrow \mathbf{GenLUT}\big(\mathtt{RightShiftSignedPaddingBlocks}(\cdot)\big)$
    $\mathsf{ct}^{(\mathsf{PaddingBlock})} \leftarrow \mathsf{PBS}\Big(\mathsf{lut}, \overrightarrow{\mathsf{ct}}_{\mathcal{B}-1}^{(\mathsf{in})}\Big)$
  else
    $\mathsf{lut} \leftarrow \mathbf{GenLUT}\Big((x) \mapsto (x \gg \mathsf{shiftBetweenBlocks}) \bmod \mathsf{msg}\Big)$
    $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}.\mathsf{append}\Big(\mathsf{PBS}\Big(\mathsf{lut}, \overrightarrow{\mathsf{ct}}_{\mathcal{B}-1}^{(\mathsf{in})}\Big)\Big)$
    $\mathsf{ct}^{(\mathsf{PaddingBlock})} \leftarrow \mathsf{ct}^{(\mathsf{Triv})}(0)$
  /* Complete with padding blocks */
  for $i \in [0, \mathsf{blockShift} - 1]$ **do**
    $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}.\mathsf{append}\big(\mathsf{ct}^{(\mathsf{PaddingBlock})}\big)$
return $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

Finally, these carries are correctly aligned with their corresponding messages and added together. Note that while it is possible to merge the bit shift operation with the first iteration of the block shift—at the cost of adding a few more PBS operations—this may not always improve performance. If the hardware cannot handle the increased concurrency, some PBS computations will be delayed, resulting in a latency similar to that of the non-merged approach.

The auxiliary functions defining the look-up tables used for merged shift bits and first block shift are given in Algorithms 50 and 51. The bit shift and bit rotation are given in Algorithm 54, with the details for the first round of the Barrel shift with encrypted power of 2 given in Algorithm 52 and the details of the subsequent rounds in Algorithm 53. The bit shift and bit rotation functions also use the block Barrel shift for the power of 2, as defined in Algorithm 43.

Algorithm 49: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathsf{RightRotate}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}, v\right)$

Input: $\begin{cases} \overrightarrow{\mathsf{ct}}^{(\mathsf{in})} : \text{homomorphic integer} \\ v \in [0, \mathcal{B} \cdot \log_2(\mathsf{msg}) - 1] \end{cases}$
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$: homomorphic integer

```
1  blockShift ← v // log2(msg)
2  shiftBetweenBlocks ← v mod log2(msg)
3  ct⃗(out) ← BlocksRotateLeft(ct⃗(in))
4  if shiftBetweenBlocks ≠ 0 then
5      ct⃗(out) ← {}
6      lut ← GenLUT((current, next) ↦ RightShiftBetweenBlocks(current, next, shiftBetweenBlocks))
7      for i ∈ [0, B − 1] do
8          next ← if i = B − 1 then 0 else i + 1
9          ct⃗(out).append(BivPBS(lut, ct⃗(out)_i, ct⃗(out)_next))
10 return ct⃗(out)
```

Algorithm 50: Auxiliary functions for LUT for power of 2 shifts (part 1)

Input: block, firstAmountBlock, previous $\in [0, \mathsf{msg} - 1]$: input domains to define the λ-functions
Output: result $\in [0, \mathsf{msg} - 1]$

```
1  Function MsgForBlock(block, firstAmountBlock, kind):
2      shiftWithinBlock ← firstAmountBlock mod log2(msg)
3      shiftToNextBlock ← (firstAmountBlock // log2(msg)) mod 2
4      switch kind do
5          case LeftShift OR LeftRotate do
6              result ← (block ≪ shiftWithinBlock) mod msg
7          case RightShift OR RightRotate do
8              result ← (block ≫ shiftWithinBlock) mod msg
9      if shiftToNextBlock = 1 then
10         result = 0
11     return result
12 Function MsgForNextBlock(previous, firstAmountBlock, kind):
13     shiftWithinBlock ← firstAmountBlock mod log2(msg)
14     shiftToNextBlock ← (firstAmountBlock // log2(msg)) mod 2
15     if shiftToNextBlock = 1 then
           /* We get the message part of the previous block */
16         switch kind do
17             case LeftShift OR LeftRotate do
18                 result = (previous ≪ shiftWithinBlock) mod msg
19             case RightShift OR RightRotate do
20                 result = (previous ≫ shiftWithinBlock) mod msg
21     else
           /* We get the carry part of the previous block */
22         switch kind do
23             case LeftShift OR LeftRotate do
24                 result = previous ≫ (log2(msg) − shiftWithinBlock)
25             case RightShift OR RightRotate do
26                 result = previous ≪ (log2(msg) − shiftWithinBlock) mod msg
27     return result
```

Algorithm 51: Auxiliary functions for LUT for power of 2 shifts (part 2)

```
   Input: block, firstAmountBlock, previous ∈ [0, msg − 1] : input domains to define the λ-functions
   Output: result ∈ [0, msg − 1]
1  Function MsgForNextNextBlock(previousPrevious, firstAmountBlock, kind):
2      shiftWithinBlock ← firstAmountBlock mod log2(msg)
3      shiftToNextBlock ← (firstAmountBlock // log2(msg)) mod 2
4      if shiftToNextBlock = 1 then
           /* We get the carry part of the previous block                                   */
5          switch kind do
6              case LeftShift OR LeftRotate do
7                  result = (previousPrevious ≫ (log2(msg) − shiftWithinBlock))
8              case RightShift OR RightRotate do
9                  result = (previousPrevious ≪ (log2(msg) − shiftWithinBlock)) mod msg
10     else
           /* Nothing reaches that block                                                   */
11         result = 0
12     return result
13 Function MsgForBlockSignedRight(block, firstAmountBlock, kind):
14     shiftWithinBlock ← firstAmountBlock mod log2(msg)
15     shiftToNextBlock ← (firstAmountBlock // log2(msg)) mod 2
16     signBitPos ← log2(msg) − 1
17     signBit ← (previous ≫ signBitPos) & 1
18     paddingBlock ← (msg − 1) · signBit
19     if shiftToNextBlock = 1 then
20         result = paddingBlock;
21     else
22         block ← (paddingBlock ≪ log2(msg)) | block;
23         result = (block ≫ shiftWithinBlock) mod msg
24     return result
25 Function MsgForNextBlockSignedRight(previous, firstAmountBlock, kind):
26     shiftWithinBlock ← firstAmountBlock mod log2(msg)
27     shiftToNextBlock ← (firstAmountBlock // log2(msg)) mod 2
28     signBitPos ← log2(msg) − 1
29     signBit ← (previous ≫ signBitPos) & 1
30     paddingBlock ← (msg − 1) · signBit
31     if shiftToNextBlock = 1 then
32         previous ← (paddingBlock ≪ log2(msg)) | previous;
33         result = (previous ≫ shiftWithinBlock) mod msg
34     else
35         result = (previous ≪ (log2(msg) − shiftWithinBlock)) mod msg
36     return result
```

Algorithm 52: Auxiliary function for first round of Barrel Shift with encrypted power of 2

Input: $\overrightarrow{\mathsf{ct}}^{(\mathsf{value})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{amount})}$: homomorphic integers
kind $\in$ {LeftShift, LeftRotate, RightShift, RightRotate}
maxNumBitsThatTellShift : generally equals to $\log_2(\log_2(\Omega))$
i.e., the number of bits to represent the maximum possible shift

Output: messagesForBlock, messagesForNextBlock, messagesForNextNextBlock = $\{\mathsf{ct}\}^{\mathcal{B}}$: lists of $\mathcal{B}$ blocks of encrypted values

Function `DoFirstRound`($\overrightarrow{\mathsf{ct}}^{(\mathsf{value})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{amount})}$, kind, maxNumBitsThatTellShift):
  offset $\leftarrow \log_2(\mathsf{msg})$
  shiftBits $\leftarrow$ `ExtractBits`($\overrightarrow{\mathsf{ct}}^{(\mathsf{amount})}$, offset, maxNumBitsThatTellShift)
  messageBitsPerBlock $\leftarrow \log_2(\mathsf{msg})$
  messages $\leftarrow \{\}$
  for $i \in [0, \mathcal{B}-1]$ **do**
    if `isSigned`$\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{value})}\right)$ AND kind = RightShift AND `i` $= \mathcal{B}-1$ **then**
      lut $\leftarrow$ **GenLUT**$\Big((\mathsf{block}, \mathsf{firstAmountBlock}) \mapsto$ `MsgForBlockSignedRight`$(\mathsf{block}, \mathsf{firstAmountBlock}, \mathsf{kind})\Big)$
    else
      lut $\leftarrow$ **GenLUT**$\Big((\mathsf{block}, \mathsf{firstAmountBlock}) \mapsto$ `MsgForBlock`$(\mathsf{block}, \mathsf{firstAmountBlock}, \mathsf{kind})\Big)$
    messages.append$\Big(\mathsf{BivPBS}\Big(\mathsf{lut}, \overrightarrow{\mathsf{ct}}^{(\mathsf{value})}, \overrightarrow{\mathsf{ct}}_0^{(\mathsf{amount})}\Big)\Big)$
  messagesForNextBlock $\leftarrow \{\}$
  switch kind **do**
    case RightShift **do**
      messagesForNextBlock.append$\big(\mathsf{ct}^{(\mathsf{Triv})}(0)\big)$
      range $\leftarrow [1, \mathcal{B}-1]$
    case LeftShift **do** range $\leftarrow [0, \mathcal{B}-2]$;
    case LeftRotateORRightRotate **do** range $\leftarrow [0, \mathcal{B}-1]$;
  for range **do**
    if `isSigned`$\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{value})}\right)$ and kind = RightShift AND $i = \mathcal{B}-1$ **then**
      LUT $\leftarrow$ **GenLUT**$\Big((\mathsf{block}, \mathsf{firstAmountBlock}) \mapsto$ `MsgForBlockSignedRight`$(\mathsf{block}, \mathsf{firstAmountBlock}, \mathsf{kind})\Big)$
    else
      lut $\leftarrow$ **GenLUT**$\Big((\mathsf{block}, \mathsf{firstAmountBlock}) \mapsto$ `MsgForBlock`$(\mathsf{block}, \mathsf{firstAmountBlock}, \mathsf{kind})\Big)$
    messages.append$\Big(\mathsf{BivPBS}\Big(\mathsf{lut}, \overrightarrow{\mathsf{ct}}^{(\mathsf{value})}, \overrightarrow{\mathsf{ct}}_0^{(\mathsf{amount})}\Big)\Big)$
  if kind = LeftShift **then**
    messagesForNextBlock.append$\big(\mathsf{ct}^{(\mathsf{Triv})}(0)\big)$
  messagesForNextNextBlock $\leftarrow \{\}$
  switch kind **do**
    case RightShift **do**
      messagesForNextNextBlock.append$\big(\mathsf{ct}^{(\mathsf{Triv})}(0)\big)$
      messagesForNextNextBlock.append$\big(\mathsf{ct}^{(\mathsf{Triv})}(0)\big)$
      range $\leftarrow [2, \mathcal{B}-1]$
    case LeftShift **do** range $\leftarrow [0, \mathcal{B}-3]$;
    case LeftRotate OR RightRotate **do** range $\leftarrow [0, \mathcal{B}-1]$;
  for $i \in$ range **do**
    lut $\leftarrow$ **GenLUT**$\Big((\mathsf{block}, \mathsf{firstAmountBlock}) \mapsto$ `MsgForBlock`$(\mathsf{block}, \mathsf{firstAmountBlock})\Big)$
    messages.append$\Big(\mathsf{BivPBS}\Big(\mathsf{lut}, \overrightarrow{\mathsf{ct}}_i^{(\mathsf{value})}, \overrightarrow{\mathsf{ct}}_0^{(\mathsf{amount})}\Big)\Big)$
  if kind = LeftShift **then**
    messagesForNextNextBlock.append$\big(\mathsf{ct}^{(\mathsf{Triv})}(0)\big)$
    messagesForNextNextBlock.append$\big(\mathsf{ct}^{(\mathsf{Triv})}(0)\big)$
  return messagesForBlock, messagesForNextBlock, messagesForNextNextBlock

Algorithm 53: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathsf{BitBarrelShiftPowerOfTwo}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{value})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{amount})}, \mathsf{kind}, \mathsf{maxBits}\right)$

Input: $\begin{cases} \overrightarrow{\mathsf{ct}}^{(\mathsf{value})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{amount})} : \text{homomorphic integers} \\ \mathsf{kind} \in \{\mathsf{LeftShift}, \mathsf{LeftRotate}, \mathsf{RightShift}, \mathsf{RightRotate}\} \\ \mathsf{maxBits} \in \mathbb{N} \end{cases}$

Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$: homomorphic integer

Function `BitBarrelShiftPowerOfTwo`$(\overrightarrow{\mathsf{ct}}^{(\mathsf{value})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{amount})}, \mathsf{kind}, \mathsf{maxBits})$**:**

- $\mathsf{messagesForBlock}, \mathsf{messagesForNextBlock}, \mathsf{messagesForNextNextBlock} \leftarrow$ `DoFirstRound`$\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{value})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{amount})}, \mathsf{kind}, \mathsf{maxBits}\right)$
- **for** $i \in [0, \mathcal{B}-1]$ **do**
  - $\mathsf{messages}_i \leftarrow \mathsf{messagesForBlock}_i + \mathsf{messagesForNextBlock}_i + \mathsf{messagesForNextNextBlock}_i$
- $\mathsf{messageBitsPerBlock} \leftarrow \log_2(\mathsf{msg})$
- $\mathsf{shiftBits} \leftarrow \mathsf{ExtractBits}(\mathsf{amount}, \mathsf{offset} = \mathsf{messageBitsPerBlock}, \mathsf{maxBits})$
- $\mathsf{numBitThatTellsShiftWithinBlocks} \leftarrow \log_2(\mathsf{messageBitsPerBlock})$
- $\mathsf{numBitsAlreadyDone} \leftarrow \mathsf{numBitThatTellsShiftWithinBlocks} + 1$
- $\mathsf{shiftBits} \leftarrow \left\{\mathsf{shiftBits}_{\mathsf{numBitsAlreadyDone}}, \cdots, \mathsf{shiftBits}_{\mathrm{len}(\mathsf{shiftBits})}\right\}$
- $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} =$ `BlockBarrelShift`$(\mathsf{messages}, \mathsf{shiftBits}, [1, \mathsf{maxBits} - \mathsf{numBitsAlreadyDone}])$
- **return** $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

Algorithm 54: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathsf{BitRotateShift}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{value})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{amount})}, \mathsf{kind}\right)$

Input: $\begin{cases} \overrightarrow{\mathsf{ct}}^{(\mathsf{value})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{amount})} : \text{homomorphic integers} \\ \mathsf{kind} \in \{\mathsf{LeftShift}, \mathsf{LeftRotate}, \mathsf{RightShift}, \mathsf{RightRotate}\} \end{cases}$

Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$: homomorphic integer

if $\mathsf{msg} = 2$ **then**

- $\mathsf{shiftBits} \leftarrow \mathsf{ExtractBits}(\mathsf{amount})$
- $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow$ `BlockBarrelShift`$(\overrightarrow{\mathsf{ct}}^{(\mathsf{value})}, \mathsf{shiftBits}, \mathsf{kind}, \lceil \log_2(\mathcal{B}) \rceil)$

else

- $n \leftarrow \lceil \log_2 \lceil \log_2(\Omega) \rceil \rceil$
- $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow$ `BitBarrelShiftPowerOfTwo`$(\overrightarrow{\mathsf{ct}}^{(\mathsf{value})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{amount})}, \mathsf{kind}, n)$

return $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

Theorem 24 states the guarantees of Algorithm 54 and its complexity.

Theorem 24 *Let $\overrightarrow{\mathsf{ct}}^{(1)}$ and $\overrightarrow{\mathsf{ct}}^{(2)}$ be two large homomorphic integers. Let $\mu = \lceil \log_2(\mathcal{B} \cdot \log_2(\mathsf{msg}))\rceil = \lceil \log_2(\lceil \log_2(\Omega)\rceil)\rceil$ be the number of bits required to represent the number of bits encrypted by $\overrightarrow{\mathsf{ct}}^{(1)}$. For example, with $\mathsf{msg} = 4$, $\mathcal{B} = 4$, the number of bits encrypted by $\overrightarrow{\mathsf{ct}}^{(1)}$ is $\mathcal{B} \cdot \log_2(\mathsf{msg}) = 8$ so we allow shifts in the range $[0, 7]$ meaning that any shift can be represented using $\lceil \log_2(\mathcal{B} \cdot \log_2(\mathsf{msg}))\rceil = 3$ bits. Thus, μ allows us to extract the appropriate amount of bits from $\overrightarrow{\mathsf{ct}}^{(2)}$ to perform the shift in the allowed range.*

Then,

$$\mathsf{Dec_s}\Big(\mathsf{LeftShift}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big)\Big) = m^{(1)} \ll (m^{(2)} \bmod \mu) = (m^{(1)} \cdot 2^{m^{(2)}}) \pmod{\Omega},$$
$$\mathsf{Dec_s}\Big(\mathsf{RightShift}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big)\Big) = m^{(1)} \gg (m^{(2)} \bmod \mu) = (m^{(1)} \,//\, 2^{m^{(2)}}) \pmod{\Omega},$$
$$\mathsf{Dec_s}\Big(\mathsf{RightRotate}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big)\Big) = (m^{(1)} \ll (m^{(2)} \bmod \mu)) \mid (m^{(1)}(\gg log_2(\Omega) - m^{(2)})),$$
$$\mathsf{Dec_s}\Big(\mathsf{LeftRotate}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\Big)\Big) = (m^{(1)} \gg (m^{(2)} \bmod \mu)) \mid (m^{(1)}(\ll log_2(\Omega) - m^{(2)})).$$

If $\mathsf{msg} > 2$, the algorithmic complexity is:

$$\mathbb{C}^{\mathsf{Shift}} = \mathbb{C}^{\mathsf{PBS}} \cdot (\mu + (\mu \cdot 2 \cdot \mathcal{B}) + \mathcal{B})$$

When using the ManyLut PBS *is possible (only for $\mathsf{msg} > 4$) we have:*

- *For left/right bit shift of unsigned integers as well as left shift of signed integers:*

$$\mathbb{C}^{\mathsf{Shift}} = \mathbb{C}^{\mathsf{PBS}} \cdot \left(\left\lceil \frac{\mu - 1 - \log_2(\mathsf{msg})}{\log_2(\mathsf{msg})} \right\rceil + \mathcal{B} + (\mathcal{B} - 1) + (\mathcal{B} - 2) + ((\mu - 1 - \log_2(\mathsf{msg}) \cdot \mathcal{B}) + \mathcal{B}\right)$$

- *For right shift of signed integers:*

$$\mathbb{C}^{\mathsf{Shift}} = \mathbb{C}^{\mathsf{PBS}} \cdot \left(\left\lceil \frac{\mu - 1 - \log_2(\mathsf{msg})}{\log_2(\mathsf{msg})} \right\rceil + \mathcal{B} + (\mathcal{B} - 1) + (\mathcal{B} - 2) + ((\mu - 1 - \log_2(\mathsf{msg}) \cdot (\mathcal{B} + 1)) + \mathcal{B}\right)$$

- *For left and right rotations, either signed or unsigned:*

$$\mathbb{C}^{\mathsf{Shift}} = \mathbb{C}^{\mathsf{PBS}} \cdot \left(\left\lceil \frac{\mu - 1 - \log_2(\mathsf{msg})}{\log_2(\mathsf{msg})} \right\rceil + (3 \cdot \mathcal{B}) + (\mu - 1 - \log_2(\mathsf{msg}))) \cdot \mathcal{B} + \mathcal{B}\right).$$

3.11 Absolute Value

The absolute value can only be applied to *signed* homomorphic integers, producing an *unsigned* homomorphic integer. In fact, this operation is redundant for unsigned integers, as they are inherently non-negative. The algorithm for computing the absolute value of a signed $\log_2(\Omega)$-bit integer m is given in Algorithm 55, and Theorem 25 states its guarantees and complexity. Informally, it consists of the following steps:

1. Perform an arithmetic right shift (cf. Remark 35) on m by $\log_2(\Omega) - 1$ bits, producing the output mask.
2. Compute $m + \mathsf{mask}$.
3. Compute an XOR between $(m + \mathsf{mask})$ and mask.

As an example, consider a signed 8-bit representation of -6, which is written 11111010_2. Performing the arithmetic right shift by 7 bits (step 1) and recalling that in the signed case, it fills the empty bits with a copy of the sign bit (here it is 1), we obtain $\mathsf{mask} = 11111111_2$. The next step is to perform addition between these two values, i.e.,

$$11111010_2 + 11111111_2 = 11111001_2,$$

which yields -7. Finally, we retrieve the absolute value by performing a bitwise XOR between -7 and mask, which yields:

$$11111001_2 \oplus 11111111_2 = 00000110_2,$$

which is 6 in the unsigned representation, as expected.

Algorithm 55: $\overrightarrow{\mathsf{ct}}^{(\mathsf{Abs})} \leftarrow \mathsf{Abs}\left(\overrightarrow{\mathsf{ct}}^{(1)}\right)$

Input: $\overrightarrow{\mathsf{ct}}^{(1)}$: a signed large homomorphic integer
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{Abs})}$: an unsigned large homomorphic integer

1 $\mathsf{numBits} \leftarrow \mathcal{B} \cdot \log_2(\mathsf{msg})$
2 $\overrightarrow{\mathsf{ct}}^{(\mathsf{mask})} = \mathtt{RightShift}(\overrightarrow{\mathsf{ct}}^{(1)}, \mathsf{numBits} - 1)$
3 $\overrightarrow{\mathsf{ct}}^{(\mathsf{sum})} = \mathsf{Add}\left(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(\mathsf{mask})}\right)$
4 $\overrightarrow{\mathsf{ct}}^{(\mathsf{Abs})} = \mathsf{Bitwise}\left(\mathsf{XOR}, \overrightarrow{\mathsf{ct}}^{(\mathsf{mask})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{sum})}\right)$
5 **return** $\overrightarrow{\mathsf{ct}}^{(\mathsf{Abs})}$

Theorem 25 *Let $\overrightarrow{\mathsf{ct}}^{(1)}$ be a signed large homomorphic integer that encrypts $m^{(1)} \in \left[-\frac{\Omega}{2}, \frac{\Omega}{2} - 1\right]$. Then,*

$$\mathsf{Dec}_{\mathsf{s}}\left(\mathsf{Abs}\left(\overrightarrow{\mathsf{ct}}^{(1)}\right)\right) = \left|m^{(1)}\right| \in [0, \Omega - 1].$$

The algorithmic complexity is:

$$\mathbb{C}^{\mathsf{Abs}} = \mathbb{C}^{\mathsf{Shift}} + \mathbb{C}^{\mathsf{Add}} + \mathbb{C}^{\mathsf{Bitwise}}$$

3.12 Sum

While summation is a useful operation on its own, it also serves as a crucial building block for other operations, such as multiplication. A naïve implementation would simply accumulate the result into a

variable and perform carry propagation after each addition. However, this approach would be highly costly and fail to fully utilize the available space in the carry buffers.

Given a list of homomorphic integers where each block encrypts a value less than msg, the expression $\frac{\mathsf{p}-1}{\mathsf{msg}-1}$ represents the maximum number of ciphertexts that can be summed before completely filling their carry buffers. Once the carry buffers are full, the next step is to extract the message part ($x \bmod \mathsf{msg}$) and the carry part ($x \mathbin{//} \mathsf{msg}$) of each block using PBS. These are then used to create two new integer ciphertexts, ($\overrightarrow{\mathsf{ct}}^{(m)}$ and $\overrightarrow{\mathsf{ct}}^{(c)}$), which can be added back to the list of integers to be summed. This process can be repeated until fewer than $\frac{\mathsf{p}-1}{\mathsf{msg}-1}$ ciphertexts remain in the list. At this point, $\overrightarrow{\mathsf{ct}}^{(m)}$ and $\overrightarrow{\mathsf{ct}}^{(c)}$ are added together and a carry propagation happens.

Blocks with $\deg = 0$ have as only value possible 0, which does not affect the final sum value, thus these blocks can be skipped. This small optimization is beneficial mainly in the multiplication to avoid redundant work as the terms have some trivial ciphertexts. To make writing the algorithm slightly simpler, the data is processed as columns of blocks as opposed to processing lines of radix ciphertexts.

Theorem 26 *Let* $\left\{\overrightarrow{\mathsf{ct}}^{(0)}, \cdots \overrightarrow{\mathsf{ct}}^{(n-1)}\right\}$ *be a list of* n *large homomorphic integers. Then,*

$$\mathsf{Dec}_{\mathsf{s}}\left(\mathsf{Sum}\left(\left\{\overrightarrow{\mathsf{ct}}^{(0)}, \cdots \overrightarrow{\mathsf{ct}}^{(\mathsf{n}-1)}\right\}\right)\right) = \sum_{i=0}^{n-1} m^{(i)} \bmod \Omega.$$

Let $S = \frac{\mathsf{p}-1}{\mathsf{msg}-1}$ *be the maximum number of ciphertexts that can be summed before completely filling their carry buffers. Let* c_i *be the number of blocks in the column at index* i*. Let* $C(x) = \left\lceil \frac{\max\{x-S,0\}}{S-1} \right\rceil$ *be a function that computes how many steps happen given a number of blocks.*

The complexity of the partial sum is:

$$\mathbb{C}^{\mathsf{PartialSum}}\left(\left\{\overrightarrow{\mathsf{ct}}^{(0)}, \cdots \overrightarrow{\mathsf{ct}}^{(n-1)}\right\}\right) = \mathbb{C}^{\mathsf{PBS}} \cdot \left(2\left(c_0 + \sum_{i=1}^{\mathcal{B}-2} C(c_i + C(c_{i-1}))\right) + C(c_{\mathcal{B}-1} + C(c_{\mathcal{B}-2}))\right).$$

The complexity of the sum is:

$$\mathbb{C}^{\mathsf{Sum}}\left(\left\{\overrightarrow{\mathsf{ct}}^{(0)}, \cdots \overrightarrow{\mathsf{ct}}^{(n-1)}\right\}\right) = \mathbb{C}^{\mathsf{PartialSum}}\left(\left\{\overrightarrow{\mathsf{ct}}^{(0)}, \cdots \overrightarrow{\mathsf{ct}}^{(n-1)}\right\}\right) + \mathbb{C}^{\mathsf{PBS}} \cdot (2\mathcal{B}) + \mathbb{C}^{\mathsf{Propagation}}.$$

Most of the time, the input of the sum is a list of n *homomorphic integers with each of them having the same number of blocks* $\mathcal{B}$*.*

We present the `Sum` algorithm in Algorithm 57, along with required subroutines in Algorithm 56. We also require usage of the `FullPropagate` algorithm, which is outlined in Algorithm 37.

Algorithm 56: Auxiliary functions used in Sum

Function `PartialSum(terms)`:
  Input: terms $= \left\{\overrightarrow{\mathsf{ct}}^{(0)}, ..., \overrightarrow{\mathsf{ct}}^{(n-1)}\right\}$: list of homomorphic integers
  Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{partialSum})}$: homomorphic integer
  numTerms $\leftarrow$ len(terms)
  /* Re-organize as columns */
  columns $\leftarrow \left\{\{\}_0, \cdots, \{\}_{\mathcal{B}-1}\right\}$
  for $i \in [0, \mathcal{B}-1]$ **do**
    for $j \in [0, \mathsf{numTerms}-1]$ **do**
      if deg(terms$[j][i]) \neq 0$ **then**
        columns$[i]$.append(terms$[j][i]$)
  /* Process columns */
  chunkSize $\leftarrow \left\lfloor \frac{\mathsf{p}-1}{\mathsf{msg}-1} \right\rfloor$
  while any(len(column) $>$ chunkSize **for** column **in** columns) **do**
    for $i \in [0, \mathcal{B}-1]$ **do**
      if len(column$[i]) \leq$ chunkSize **then** continue
      numChunks $\leftarrow$ len(column$[i]$) // chunkSize
      for $k \in [0, \mathsf{numChunks}-1]$ **do**
        sum $\leftarrow \sum_{j=0}^{\mathsf{chunkSize}-1}$ columns$[i]$.pop()
        $m, c \leftarrow (\mathsf{PBS}((x) \mapsto x \bmod \mathsf{msg}, \mathsf{sum}), \mathsf{PBS}((x) \mapsto x \mathbin{//} \mathsf{msg}, \mathsf{sum}))$
        columns$[i]$.append(m)
        if $i < \mathcal{B}-1$ **then** columns$[i+1]$.append(c)
  /* Go back to radix form */
  $\overrightarrow{\mathsf{ct}}^{(\mathsf{partialSum})} \leftarrow \{\}$
  for $i \in [0, \mathcal{B}-1]$ **do**
    $\overrightarrow{\mathsf{ct}}^{(\mathsf{partialSum})}$.append$\left(\sum_{j=0}^{\mathrm{len}(\mathsf{columns}[i])-1} \mathsf{columns}[i][j]\right)$
  return $\overrightarrow{\mathsf{ct}}^{(\mathsf{partialSum})}$

Algorithm 57: $\overrightarrow{\mathsf{ct}} \leftarrow \mathsf{Sum}\left(\left\{\overrightarrow{\mathsf{ct}}^{(0)}, \cdots, \overrightarrow{\mathsf{ct}}^{(n-1)}\right\}\right)$

Input: $\left\{\overrightarrow{\mathsf{ct}}^{(0)}, \cdots, \overrightarrow{\mathsf{ct}}^{(n-1)}\right\}$: list of homomorphic integers
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{Sum})}$: homomorphic integer
return `FullPropagate`$\left(\mathtt{PartialSum}\left(\left\{\overrightarrow{\mathsf{ct}}^{(0)}, \cdots, \overrightarrow{\mathsf{ct}}^{(n-1)}\right\}\right)\right)$

3.13 Multiplication

Multiplication is a core operation on homomorphic integers. Below, we present two types of multiplication: multiplication between two homomorphic integers and multiplication between a homomorphic integer and a plaintext. Both implementations of the multiplication only compute the *half multiplication*, that is, for a $\mathcal{B}$-by-$\mathcal{B}$ multiplication, the output has $\mathcal{B}$ blocks.

3.13.1 Ciphertext-Ciphertext Multiplication

The ciphertext-ciphertext multiplication (Algorithm 58) uses a standard schoolbook algorithm. This algorithm can be understood as having two phases: (1) the creation of so-called *terms* and (2) the summation of these terms. The *term* creation phase involves numerous PBSes to compute block-by-block (digit-by-digit) multiplications. Since the product of two blocks exceeds msg, it would occupy

part of the carry space in a way that prevents summation of the blocks (assuming $\mathsf{msg} = \mathsf{carry}$). Thus, the complete result of the multiplication of two blocks is done using two bivariate PBSes to get the result directly split in to two blocks with the functions:

$$\mathsf{messageOfBlockMul}(x, y) \mapsto (x \cdot y) \bmod \mathsf{msg},$$
$$\mathsf{carryOfBlockMul}(x, y) \mapsto (x \cdot y) \mathbin{//} \mathsf{msg}.$$

For a $\mathcal{B}_1$-by-$\mathcal{B}_2$ multiplication, we obtain $\mathcal{B}_1 \cdot \mathcal{B}_2$ ciphertexts from bivariate PBSes. The **Sum** algorithm (Algorithm 57) can later take advantage of the empty carry buffers in these ciphertexts.

Algorithm 58: $\overrightarrow{\mathsf{ct}}^{(\mathsf{Mul})} \leftarrow \mathsf{Mul}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{lhs})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{rhs})}\right)$

Input: $\overrightarrow{\mathsf{ct}}^{(\mathsf{lhs})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{rhs})}$: two homomorphic integers
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$: a homomorphic integer

terms $\leftarrow \{\}$
for $i \in [0, \mathcal{B} - 1]$ **do**
  if $\deg\left(\mathsf{ct}_i^{(\mathsf{rhs})}\right) = 0$ **then** continue
  term $\leftarrow \left\{\mathsf{ct}^{(\mathsf{Triv})}(0)\right\}^i$
  for $j \in [0, \mathcal{B} - i - 1]$ **do**
    term.append$\left(\mathsf{BivPBS}\left(\mathsf{messageOfBlockMul}(\cdot,\cdot), \mathsf{ct}_i^{(\mathsf{rhs})}, \mathsf{ct}_j^{(\mathsf{lhs})}\right)\right)$
  terms.append(term)
if msg > 2 **then**
  for $i \in [0, \mathcal{B} - 2]$ **do**
    if $\deg\left(\mathsf{ct}_i^{(\mathsf{rhs})}\right) = 0$ **then** continue
    term $\leftarrow \left\{\mathsf{ct}^{(\mathsf{Triv})}(0)\right\}^{i+1}$
    for $j \in [0, \mathcal{B} - i - 2]$ **do**
      term.append$\left(\mathsf{BivPBS}\left(\mathsf{carryOfBlockMul}(\cdot,\cdot), \mathsf{ct}_i^{(\mathsf{rhs})}, \mathsf{ct}_j^{(\mathsf{lhs})}\right)\right)$
    terms.append(term)
return Sum(terms)

Theorem 27 states the guarantees and complexity of Algorithm 58.

Theorem 27 *Let $\overrightarrow{\mathsf{ct}}^{(1)}$ and $\overrightarrow{\mathsf{ct}}^{(2)}$ be two large homomorphic integers. Then the output of Algorithm 58 satisfies:*

$$\mathsf{Dec}_{\mathsf{s}}\left(\mathsf{Mul}\left(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}\right)\right) = \mathsf{Dec}_{\mathsf{s}}\left(\overrightarrow{\mathsf{ct}}^{(1)}\right) \cdot \mathsf{Dec}_{\mathsf{s}}\left(\overrightarrow{\mathsf{ct}}^{(2)}\right) \bmod \Omega.$$

The algorithmic complexity is:

$$\mathbb{C}^{\mathsf{Multiplication}} = \mathbb{C}^{\mathsf{PBS}} \cdot \mathcal{B}_1 \cdot \mathcal{B}_2 + \mathbb{C}^{\mathsf{PartialSum}}\left(\left\{1, 2, \cdots, (1 + 2^{\mathcal{B}})\right\}\right) + \mathbb{C}^{\mathsf{PBS}} \cdot (2\mathcal{B} - 3) + \mathbb{C}^{\mathsf{Propagation}}(\mathcal{B} - 2)$$

3.13.2 Plaintext-Ciphertext Multiplication

The plaintext-ciphertext multiplication uses the same principle as the standard binary multiplication recalled in Algorithm 59 and is presented in Algorithm 60.

Algorithm 59: Binary Multiplication

Input: $x, y \in \mathbb{Z}$
Output: $x \cdot y \in \mathbb{Z}$

```
result ← 0
for i ∈ [0, numBitsOf(y) − 1] do
    if y & 1 = 1 then
        result ← result + x
    y ← y ≫ 1
    x ← x ≪ 1
return result
```

The algorithm leverages the fact that the scalar is a known value; it means that we can decompose it into bits and use them as decision bits in an if statement. Another trick is that the standard binary multiplication of $x \cdot y$ essentially creates a list of all possible shifted values of x: `allShifts = [x ≪ i for i in range(nbBits(x))]` and then uses the bits of y to select whether a particular shift value has to be added to the partial sum. In the case of our radix representation, `allShifts` can be computed more efficiently than calling our `LeftShift` multiple times for all needed values.

In fact, we have for a shift s that $\mathsf{shiftAmount} = (s \bmod \log_2(\mathsf{msg})) + k \cdot \log_2(\mathsf{msg})$ for some k. Thus, it suffices to compute all the shifts for $s \in [0, \log_2(\mathsf{msg}) - 1]$, then only a `BlocksShiftLeft` needs to be applied to get any desired shift.

Algorithm 60: $\overrightarrow{\mathsf{ct}}^{(\mathsf{ScalarMul})} \leftarrow \mathsf{ScalarMul}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{lhs})}, \upsilon\right)$

Input: $\begin{cases} \overrightarrow{\mathsf{ct}}^{(\mathsf{lhs})}: \text{ a homomorphic integer} \\ \upsilon \in \mathbb{Z}_\Omega \end{cases}$
Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{ScalarMul})}:$ a homomorpic integer

```
preComputedShifts ← {}
for bitShift ∈ [0, log2(msg)] − 1 do
    preComputedShifts.append(LeftShift(ct^(lhs), bitShift))
terms ← {}
bitLen ← numBitsOf(υ)
for i ∈ [0, bitLen − 1] do
    if (υ & 1) = 1 then
        bitShift, blockShift ← i mod log2(B), i // log2(B)
        terms.append(BlocksShiftRight(preComputedShifts[bitShift], blockShift))
    υ ← υ ≫ 1;
return Sum(terms)
```

3.14 Detecting Overflows

When a user calls an operation with overflow detection, an additional (encrypted) boolean is returned alongside the result computation. This boolean represents whether an overflow is present or not, and therefore allows a user to detect overflows in their computation flow. In this section, we consider three cases: addition in Section 3.14.1, subtraction in Section 3.14.2, and multiplication in Section 3.14.3.

3.14.1 Addition

The method to detect overflows is different for signed and unsigned integers. The computations to determine if an overflow occurs can be performed during the carry propagation (Section 3.5) as the carry is needed to know if an overflow occurred. For both signed and unsigned integers, the overflow check

computation adds 2 PBSes and these can be scheduled to be done in parallel to other computations of the carry propagation, meaning that the latency is not degraded assuming the hardware can handle this extra added concurrency.

Unsigned. For unsigned the idea is simple, it suffices to compute the carry from the most significant block of the homomorphic integer. The way it is done is that after the first step (Section 3.5.1) the block that encodes the state of the most significant block $\overrightarrow{\mathsf{ct}}_{\mathcal{B}-1}^{(\mathsf{blockStates})}$ is kept instead of being discarded, this block is called the overflow block and currently encrypts a value in $\{0, 2, 4\}$ depending on if the last block is *neither*, *propagate* or *generate*, respectively. After the propagation states have been computed in the second step (Section 3.5.2), the last propagation state $\overrightarrow{\mathsf{ct}}_{\mathcal{B}-1}^{(\mathsf{propStates})}$ is added (using the regular LWE addition) to the overflow block. Once the group carries have been resolved in step three (Section 3.5.3), the last group carry, in other words, the carry that the last group takes as input, is also added to the overflow block (this will be taken as a trivial encryption of zero if there are no group carries). Lastly, while the final step (Section 3.5.4) is being performed, a PBS is done on the overflow block with the following function:

$$(x) \mapsto (x \gg 2) \,\&\, 1.$$

The result of this PBS is the overflow flag, a ciphertext that encrypts 0 if there was no overflow or 1 if there was an overflow, using the Boolean encoding (Section 2.3.2).

Signed. For signed integers, as they use the two's complement, the *overflow flag* is computed. This flag is obtained by doing an XOR between the input carry to the last bit and the output carry of the last bit, which can be achieved by a PBS. In case the blocks encrypt more than one bit ($\mathsf{msg} > 2$), some adaptations need to be made.

The computation of the overflow flag needs to be done in two steps, both of which can be executed in parallel to other computations.

The first overflow flag step consists in computing a ciphertext, called the overflow block, that will hold a value that, when later combined with the input carry to the last block, will allow us to conclude on the overflow. It can be done in parallel with the step where the block states are computed (Section 3.5.1).

To compute the overflow block, a bivariate PBS is done between the most significant blocks of both inputs, the function used to generate the lookup table is given in Algorithm 61. This PBS will result in a ciphertext that stores the 2 possible outcomes for the overflow depending on the input carry which is still unknown at this time.

Algorithm 61: Signed overflow preparation

Function `OverflowGivenCarry`(lhs, rhs, c):
  Input: $\begin{cases} \mathsf{lhs}, \mathsf{rhs} \in [0, \mathsf{msg}-1]^2 \text{: The possible values of most significant blocks of signed integers} \\ c \in \{0,1\} \text{: The carry received} \end{cases}$
  Output: 1 if there is an overflow, 0 otherwise
  $\mathsf{mask} \leftarrow (\mathsf{msg} \mathbin{//} 2) - 1$
  $\mathsf{lhs}_{\mathsf{low}} \leftarrow \mathsf{lhs} \,\&\, \mathsf{mask}$
  $\mathsf{rhs}_{\mathsf{low}} \leftarrow \mathsf{rhs} \,\&\, \mathsf{mask}$
  $\mathsf{outputCarry} \leftarrow (\mathsf{lhs} + \mathsf{rhs} + c) \mathbin{//} \mathsf{msg}$
  $\mathsf{inputCarryToLastbit} \leftarrow (\mathsf{lhs}_{\mathsf{low}} + \mathsf{rhs}_{\mathsf{low}} + c) \mathbin{//} (\mathsf{msg} \mathbin{//} 2)$
  return $\mathsf{outputCarry} \neq \mathsf{inputCarryToLastbit}$

Function `OverflowFlagPreparation`(lhs, rhs):
  Input: $\mathsf{lhs}, \mathsf{rhs} \in [0, \mathsf{msg}-1]^2$: The possible values of most significant blocks of signed integers
  Output: 1 if there is an overflow, 0 otherwise
  $b_0 \leftarrow$ `OverflowGivenCarry`(lhs, rhs, 1)
  $b_1 \leftarrow$ `OverflowGivenCarry`(lhs, rhs, 0)
  return $(b_1 \ll 3) \mid (b_0 \ll 2)$

After the propagation states have been computed in the second step (Section 3.5.2), the last propagation state is added (using the regular LWE addition) to the overflow block.

Once the group carries have been resolved in the third step (Section 3.5.3), the last carry, in other words the carry that the last group takes as input is added to the overflow block.

Lastly, while final step (Section 3.5.4) is being done, a PBS is done on the overflow flag with the following function used to generate the lookup table:

Algorithm 62: Signed overflow finalization

Function `OverflowFlagFinalization`(x):
  Input: $x \in [0, \mathsf{msg}-1]$: The possible value of most significant blocks of signed integer
  Output: 1 if there is an overflow, 0 otherwise
  $\mathsf{inputCarry} \leftarrow (x \gg 1) \,\&\, 1$
  $\mathsf{doesOverflowIfCarryIsOne} \leftarrow (x \gg 3) \,\&\, 1$
  $\mathsf{doesOverflowIfCarryIsZero} \leftarrow (x \gg 2) \,\&\, 1$
  if $\mathsf{inputCarry} = 1$ **then**
    return doesOverflowIfCarryIsOne
  else
    return doesOverflowIfCarryIsZero

3.14.2 Subtraction

Signed. For *signed integers*, subtraction is done by adding the negated term, then the carry propagation follows. The overflow detection is the same as for the addition of signed integers.

Unsigned. To get the overflow alongside the subtraction of *unsigned integers*, we instead use subtraction with borrow for which the borrow of the last block indicates the overflow.

To achieve subtraction with borrow, normally each pair of blocks is subtracted, however, these subtractions might themselves underflow, setting the padding bit value to 1 which needs to be avoided. Thus, a *correcting term* of the value msg is added.

Given two blocks a and b that encrypt a value in $[0, \mathsf{msg}-1]$, the result of subtraction is in $[-(\mathsf{msg}-1), \mathsf{msg}-1]$, therefore, by adding msg, we find the final result in $[1, 2 \cdot \mathsf{msg}-1]$. This allows

us to apply a PBS on each block to get the *borrow state*:

$$\mathsf{unshifted\_state}(x) = \begin{cases} \mathsf{Borrows} & \text{if } x < \mathsf{msg}, \\ \mathsf{Propagates} & \text{if } x = \mathsf{msg}, \\ \mathsf{Neither} & \text{otherwise}, \end{cases}$$

where the values for Borrows, Propagates, Neither values are described in Section 3.5 (Generate becomes Borrows), as the borrow propagation follows a similar pattern.

As with the carry propagation (Eq. (30)), the borrows computed need to be shifted to the correct position depending on the block's group. The cleanAndShift function changes slightly to only be a shift function: $\mathsf{Shift}(x) = x \ll 1$. That is because the extra bit above msg will serve as an overflow protection when the compute borrow may later be subtracted from the block.

Now that we have the *block states* we can use the same algorithm described in the second phase of the carry propagation (Section 3.5.2) with however one difference for how *prepared blocks* are created.

In the carry propagation, we took the *shifted blocks* and added the *propagation states*, i.e. $\mathsf{prepared\_block}_i = \mathsf{shifted\_blocks}_i + \mathsf{propagation\_states}_i$. This works as the *propagation states* are seen as carries whereas in the subtraction with borrow we see them as borrows, thus in the subtraction with borrow, the formula changes to:

$$\mathsf{prepared\_block}_i = \mathsf{shifted\_blocks}_i - \mathsf{propagation\_states}_i + 1$$

The way it works is that the equation above is equivalent to

$$\mathsf{prepared\_block}_i = \mathsf{shifted\_blocks}_i + (-\mathsf{propagation\_states}_i + 1).$$

As $\mathsf{propagation\_states}_i \in \{0, 1, 2\}$ we have that $-\mathsf{propagation\_states}_i + 1 \in \{-1, 0, 1\}$ more specifically:

- 2: Generates becomes -1, meaning we will properly subtract a borrow from the next block
- 1: Propagates becomes 0, meaning if a borrow (-1) comes from a previous block, it will get propagated as $0 - 1 = -1$
- 0: Neither (Absorb) becomes 1, meaning if a borrow (-1) it will be absorbed as $1 - 1 = 0$

Once that is done the borrow between groups (which is still encoded in the $\{0, 1, 2\}$ set) can be propagated using the same methods described for the carry propagation (Section 3.5.3)

Finally the last step (Section 3.5.4) is also similar, the change is that the borrow is subtracted (as opposed to the carry which is being added). To get the overflow indicator, the borrow out of the last block must be computed.

3.14.3 Multiplication

TFHE-rs supports an overflowing variant of multiplication, where we apply similar techniques as discussed in Section 3.14.2. Full details will be given in a later version of the document.

3.15 Conditionals

One of the main differences between FHE computation and classical computation is that conditional statements cannot be used to dynamically trim computations based on encrypted values. Instead, every possible branch must be executed before selecting the correct one. While FHE allows for conditional execution, all branches must be computed in parallel, after which the appropriate result is selected,

allowing computation to proceed. The PBS provides an efficient mechanism to select the correct branch based on an encrypted condition, enabling support for conditional statements such as `if` and `if-else`.

Algorithm 63 details how to perform homomorphic `if-else` statements, while Theorem 28 states its guarantees and complexity. Note that Algorithm 63 also supports `if` statements, provided they are rewritten as `if-else` statements before calling the algorithm. In Algorithm 63, the last PBS with the `identity` function is there to decrease the noise of the output of the algorithm.

Algorithm 63: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})} \leftarrow \mathsf{Select}\left(\mathsf{ct}^{(\mathsf{condition})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{true})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{false})}\right)$

Input: $\begin{cases} \mathsf{ct}^{(\mathsf{condition})} : \text{encrypted boolean} \\ \overrightarrow{\mathsf{ct}}^{(\mathsf{true})} : \text{homomorphic integer} \\ \overrightarrow{\mathsf{ct}}^{(\mathsf{false})} : \text{homomorphic integer} \end{cases}$

Output: $\begin{cases} \overrightarrow{\mathsf{ct}}^{(\mathsf{out})} : \text{homomorphic integer that encrypts the same value as } \overrightarrow{\mathsf{ct}}^{(\mathsf{true})} \text{ if } m^{(\mathsf{cond})} = \mathsf{true}, \\ \text{or } \overrightarrow{\mathsf{ct}}^{(\mathsf{false})} \text{ if } m^{(\mathsf{cond})} = \mathsf{false} \end{cases}$

1 $\mathsf{ct}^{(\mathsf{invertedCondition})} \leftarrow \mathsf{BitwiseNOT}\left(\mathsf{ct}^{(\mathsf{condition})}\right)$
2 $\mathsf{lut} \leftarrow \mathbf{GenLUT}\left((\mathsf{cond}, \mathsf{value}) \mapsto \mathsf{cond} \cdot \mathsf{value}\right)$
3 **for** $i \in [0, \mathcal{B} - 1]$ **do**
4 $\quad \mathsf{ct}^{(\mathsf{t})} \leftarrow \mathsf{BivPBS}\left(\mathsf{lut}, \mathsf{ct}^{(\mathsf{condition})}, \mathsf{ct}_i^{(\mathsf{true})}\right)$
5 $\quad \mathsf{ct}^{(\mathsf{f})} \leftarrow \mathsf{BivPBS}\left(\mathsf{lut}, \mathsf{ct}^{(\mathsf{invertedCondition})}, \mathsf{ct}_i^{(\mathsf{false})}\right)$
6 $\quad \mathsf{ct}_i^{(\mathsf{result})} \leftarrow \mathsf{PBS}\left(\mathsf{identity}, \mathsf{ct}^{(\mathsf{t})} + \mathsf{ct}^{(\mathsf{f})}\right)$
7 **return** $\overrightarrow{\mathsf{ct}}^{(\mathsf{result})}$

Theorem 28 *Let* $\overrightarrow{\mathsf{ct}}^{(\mathsf{true})}$ *and* $\overrightarrow{\mathsf{ct}}^{(\mathsf{false})}$ *be two large homomorphic integers, and* $\mathsf{ct}^{(\mathsf{condition})}$ *an LWE ciphertext with* $\mathsf{Dec}_{\mathsf{s}}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{condition})}\right) \in \{0, 1\}$

$$\mathsf{Dec}_{\mathsf{s}}\left(\mathsf{Select}\left(\mathsf{ct}^{(\mathsf{condition})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{true})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{false})}\right)\right) = \begin{cases} \mathsf{Dec}_{\mathsf{s}}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{true})}\right), & \text{if } \mathsf{Dec}_{\mathsf{s}}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{condition})}\right) = 1 \\ \mathsf{Dec}_{\mathsf{s}}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{false})}\right), & \text{if } \mathsf{Dec}_{\mathsf{s}}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{condition})}\right) = 0 \end{cases}$$

The algorithmic complexity is: $\mathbb{C}^{\mathsf{Select}} = 3 \cdot \mathcal{B} \cdot \mathbb{C}^{\mathsf{PBS}}$

3.16 Comparisons

Comparison algorithms ($<, \leq, >, \geq$) in TFHE-rs are based on the subtraction algorithm. In particular, to perform a comparison we essentially perform a subtraction with borrow, and propagate the borrow to compute the result.

For unsigned integers, computing the overflowing subtraction (Algorithms 40 and 61) is sufficient, as the overflow flag directly corresponds to the result of the comparison. Specifically, $a - b$ overflows if and only if $a < b$. For signed integers, it is slightly less direct, but the algorithm still relies on subtraction. However, for both variants, comparison uses fewer PBSes than the full borrow/carries propagation algorithm, as only the final borrow is needed (unlike the actual result of subtraction).

There are four supported comparison operations, each of which can be evaluated using overflowing subtraction:

- $a < b \iff \mathsf{overflows}(a - b)$,
- $a \leq b \iff \neg(b < a) \iff \neg(\mathsf{overflows}(b - a))$,
- $a > b \iff b < a \iff \mathsf{overflows}(b - a)$,

- $a \geq b \iff b \leq a \iff \neg(\mathsf{overflows}(a-b))$.

As described above, $(a < b)$ is determined by computing the overflow of $(a - b)$. Similarly, $(a \leq b)$ can be expressed as $\neg(b < a)$, meaning that we can use the same approach with an additional NOT operation. The same principle applies to $(a > b)$ and $(a \geq b)$ by swapping the order of the operands. This ensures that all comparisons follow the same algorithm, differing only in whether the inputs are swapped and/or the boolean result is negated. The complete comparison algorithm for both signed and unsigned integers is presented in Algorithm 69. We now briefly outline the differences between the unsigned and signed cases.

Unsigned. As outlined, in the unsigned case, computing the subtraction overflow flag is enough. Moreover, computing the overflow flag for unsigned integer subtraction is as simple as computing the output borrow of the last block. The borrow state is computed using the function `ComputeGroupBorrowState` outlined in Algorithm 65.

Signed. For signed integers, the algorithm is more complex: both the overflow flag and the resulting sign bit have to be computed and XORed together, which requires additional PBSes compared to the unsigned case. This additional step in Algorithm 69 can be seen in the `PrepareSignedCheck` function in Algorithm 67. However, when blocks encrypt 4 bits or more (i.e., $\mathsf{p} \geq 16$), both these computations can be merged into one in a way similar to the overflow detection in case of the addition of signed numbers. This idea is shown in the `SignedCmpResultPreparation` function in Algorithm 64.

Algorithm 64: Comparison: auxiliary functions to generate LUTs specific to signed integers

Function `IsLessThanGivenCarry`$(\mathsf{lhs}, \mathsf{rhs}, c)$**:**
  Input: $\begin{cases} \mathsf{lhs} \in [0, \mathsf{msg}-1] \\ \mathsf{rhs} \in [0, \mathsf{msg}-1] \\ c \in \{0,1\} \end{cases}$
  Output: $\{0,1\}$
  $\mathsf{mask} \leftarrow (\mathsf{msg} \,//\, 2) - 1$
  $\mathsf{lhs}_{\mathsf{low}} \leftarrow \mathsf{lhs} \,\&\, \mathsf{mask}$
  $\mathsf{rhs}_{\mathsf{low}} \leftarrow \mathsf{rhs} \,\&\, \mathsf{mask}$
  $\mathsf{outputBorrow} \leftarrow \mathsf{lhs} < (\mathsf{rhs} + c)$
  $\mathsf{inputBorrowToLastBit} \leftarrow \mathsf{lhs}_{\mathsf{low}} < (\mathsf{rhs}_{\mathsf{low}} + c)$
  $r \leftarrow \mathsf{lhs} - \mathsf{rhs} + \mathsf{msg}$
  $\mathsf{signBit} \leftarrow r \,//\, (\mathsf{msg} \,//\, 2)$
  $\mathsf{overflowFlag} \leftarrow \mathsf{inputBorrowToLastBit} \oplus \mathsf{outputBorrow}$
  return $\mathsf{overflowFlag} \oplus \mathsf{signBit}$
Function `SignedCmpResultPreparation`$(\mathsf{lhs}, \mathsf{rhs})$**:**
  $b_1 \leftarrow$ `IsLessThanGivenCarry`$(\mathsf{lhs}, \mathsf{rhs}, 1)$
  $b_0 \leftarrow$ `IsLessThanGivenCarry`$(\mathsf{lhs}, \mathsf{rhs}, 0)$
  return $(b_1 \ll 3) \mid (b_0 \ll 2)$

Theorem 29 *Let $\overrightarrow{\mathsf{ct}}^{(1)}$ and $\overrightarrow{\mathsf{ct}}^{(2)}$ be two large homomorphic integers, and let $\boxplus$ denote an arbitrary condition in $\{<, \leq, >, \geq\}$. Then the output of Algorithm 69 satisfies:*

$$\mathsf{Dec}_{\mathsf{s}}\Big(\mathsf{Compare}\Big(\overrightarrow{\mathsf{ct}}^{(1)}, \overrightarrow{\mathsf{ct}}^{(2)}, \boxplus\Big)\Big) = \begin{cases} 1 & \text{if } \mathsf{Dec}_{\mathsf{s}}\Big(\overrightarrow{\mathsf{ct}}^{(1)}\Big) \boxplus \mathsf{Dec}_{\mathsf{s}}\Big(\overrightarrow{\mathsf{ct}}^{(2)}\Big), \\ 0 & \text{otherwise.} \end{cases}$$

Let $\mathsf{groupSize} = \log_2(\mathsf{p})$ *and* $\mathsf{numGroup} = \left\lceil \frac{\mathcal{B}}{\mathsf{groupSize}} \right\rceil$. *For unsigned comparisons, the complexity is:*

$$\mathbb{C}^{\mathsf{UnsignedCmp}} = (\mathcal{B} + \mathsf{numGroup} + \mathsf{numGroup} - 1) \cdot \mathbb{C}^{\mathsf{PBS}},$$

for signed comparisons, the complexity is:

$$\mathbb{C}^{\mathsf{SignedCmp}} = ((\mathcal{B}+1) + (\mathsf{numGroup}+1) + \tau) \cdot \mathbb{C}^{\mathsf{PBS}},$$

where $\tau = \begin{cases} 1 & \textit{if } \mathsf{numGroup} = 1, \\ \mathsf{numGroup} - 1 & \textit{otherwise.} \end{cases}$

Remark 37 (Plaintext-Ciphertext Comparison) *To perform a plaintext-ciphertext comparison, we can use the same algorithm by treating the plaintext as a trivial ciphertext (see Definition 6). A more efficient version is available in TFHE-rs and will be included in future versions of this document.*

Algorithm 65: Comparison: auxiliary function to compute the group borrow state

Function `ComputeGroupBorrowState(invertResult, groupSize, blockStates):`

Input: $\begin{cases} \mathsf{invertResult} \in \{\mathsf{true}, \mathsf{false}\} \\ \mathsf{groupSize} \in \mathbb{N} \\ \mathsf{blockStates} = \left(\mathsf{ct}_j^{(\mathsf{blockState})}\right)_{j=0}^{\mathcal{B}-1} : \text{a } \mathcal{B}\text{-tuple of block states} \end{cases}$

Output: $\mathsf{groupPropagationStates} = \left(\mathsf{ct}_i^{(\mathsf{groupState})}\right)_{i=0}^{\mathsf{ng}-1}$: a $\mathsf{ng} = \left\lceil \frac{\mathcal{B}}{\mathsf{groupSize}} \right\rceil$-tuple of group states

$\mathsf{numGroups} \leftarrow \left\lceil \frac{\mathcal{B}}{\mathsf{groupSize}} \right\rceil$

$\mathsf{numBlocksInLastGroup} \leftarrow \mathcal{B} \bmod \mathsf{groupSize}$

$\mathsf{numCarries} \leftarrow \mathsf{numGroups} - 1$

$\mathsf{seqDepth} \leftarrow (\mathsf{numCarries} - 1) \;//\; (\mathsf{groupSize} - 1)$

$\mathsf{hillisSteeleDepth} \leftarrow$ **if** $\mathsf{numCarries} = 0$ **then** 0 **else** $\lceil \log_2(\mathsf{numCarries}) \rceil$

$\mathsf{useSeq} \leftarrow \mathsf{seqDepth} \leq \mathsf{hillisSteeleDepth}$

/* The last group may not be full */

if $\mathsf{numGroups} = 1$ **then**

  $\mathsf{lastGroupLut} \leftarrow \mathbf{GenLUT}\Big((x) \mapsto ((x \gg (\mathsf{numBlocksInLastGroup} - 1)) \,\&\, 1) \oplus \mathsf{invertResult}\Big)$

  $\mathsf{lastGroupLutCorrector} \leftarrow 0$

else

  /* There are at least 2 groups, the first and the last having specific LUTs */

  $\mathsf{firstGroupLut} \leftarrow \mathbf{GenLUT}\Big((x) \mapsto (x \gg (\log_2(\mathsf{p}) - 1)) \,\&\, 1\Big)$

  $\mathsf{otherGroupLuts} \leftarrow \{\}$

  if useSeq **then**

    for $i \in [0, \mathsf{groupSize} - 2]$ **do**

      $\mathsf{otherGroupLuts.append}\Big(\mathbf{GenLUT}\Big((x) \mapsto (-(x \neq \mathsf{p} - 1) \ll i) \bmod 2\mathsf{p}\Big)\Big)$

  else

    Function `LocalFn(`x`):`

      if $x = \mathsf{p} - 1$ **then return** 2 **else return** -1

    $\mathsf{otherGroupLuts.append}\big(\mathbf{GenLUT}\big(\texttt{LocalFn}(\cdot)\big)\big)$

  if $\mathsf{numBlocksInLastGroup} = \mathsf{groupSize}$ **then**

    $\mathsf{lastGroupLut} \leftarrow \mathsf{otherGroupLuts}[0]$

    $\mathsf{lastGroupLutCorrector} \leftarrow 1$

  else

    Function `LocalFn(`x`):`

      if $x \geq (1 \ll \mathsf{numBlocksInLastGroup})$ **then**

        return 2

      else if $x = (1 \ll \mathsf{numBlocksInLastGroup}) - 1$ **then**

        return 1

      else

        return 0

    $\mathsf{lastGroupLut} \leftarrow \mathbf{GenLUT}\big(\texttt{LocalFn}(\cdot)\big)$

    $\mathsf{lastGroupLutCorrector} \leftarrow 0$

$\mathsf{groupPropagationStates} \leftarrow \{\}$

for $i \in [0, \mathsf{numGroups} - 1]$ **do**

  $\mathsf{sum} \leftarrow \sum_{j=i\cdot\mathsf{groupSize}}^{\min(\mathcal{B}-1,(i+1)\cdot\mathsf{groupSize})} \mathsf{ct}_j^{(\mathsf{blockState})}$

  if $i = \mathsf{numGroups} - 1$ **then**

    $\mathsf{result} \leftarrow \mathsf{PBS}(\mathsf{lastGroupLut}, \mathsf{sum})$

    $\mathsf{result} \leftarrow \mathsf{result} + \mathsf{lastGroupLutCorrector}$

  else if $i = 0$ **then**

    $\mathsf{result} \leftarrow \mathsf{PBS}(\mathsf{firstGroupLut}, \mathsf{sum})$

  else

    $\mathsf{index} \leftarrow$ **if** useSeq **then** $(i - 1) \bmod (\mathsf{groupSize} - 1)$ **else** 0

    $\mathsf{result} \leftarrow \mathsf{PBS}(\mathsf{otherGroupLuts}[\mathsf{index}], \mathsf{sum})$

    $\mathsf{corrector} \leftarrow$ **if** useSeq **then** $1 \ll ((i - 1) \bmod (\mathsf{groupSize} - 1))$ **else** 1

    $\mathsf{result} \leftarrow \mathsf{result} + \mathsf{corrector}$

  $\mathsf{groupPropagationStates.append}(\mathsf{result})$

return groupPropagationStates

Algorithm 66: Comparison: auxiliary functions to compare two values given a borrow

Function `IsXLessThanYGivenInputBorrow`(lastXBlock, lastYBlock, borrow):

Input: $\begin{cases} \mathsf{lastXBlock} \in [0, \mathsf{msg}-1] \\ \mathsf{lastYBlock} \in [0, \mathsf{msg}-1] \\ \mathsf{borrow} \in \{0,1\} \end{cases}$

Output: result $\in \{\mathsf{true}, \mathsf{false}\}$

lastBitPos $\leftarrow \log_2(\mathsf{msg}) - 1$
mask $\leftarrow (1 \ll \mathsf{lastBitPos}) - 1$
xWithoutLastBit $\leftarrow$ lastXBlock $\&$ mask
yWithoutLastBit $\leftarrow$ lastYBlock $\&$ mask
inputBorrowToLastBit $\leftarrow$ xWithoutLastBit $<$ (yWithoutLastBit + borrow)
result $\leftarrow$ lastXBlock + lastYBlock + borrow
outputSignBit $\leftarrow$ (result $\gg$ lastBitPos) $\&$ 1
outputBorrow $\leftarrow$ lastXBlock $<$ (lastYBlock + borrow)
overflowFlag $\leftarrow$ inputBorrowToLastBit $\oplus$ outputBorrow
return outputSignBit $\oplus$ overflowFlag

Algorithm 67: Comparison: auxiliary functions to prepare the signed check

Function `PrepareSignedCheck`($\mathsf{ct}^{(\mathsf{lhs})}_{\mathcal{B}-1}, \mathsf{ct}^{(\mathsf{rhs})}_{\mathcal{B}-1}$):

Input: $\begin{cases} \mathsf{ct}^{(\mathsf{lhs})}_{\mathcal{B}-1}\text{: the most significant block of a signed homomorphic integer} \\ \mathsf{ct}^{(\mathsf{rhs})}_{\mathcal{B}-1}\text{: the most significant block of a signed homomorphic integer} \end{cases}$

Output: $\begin{cases} \mathsf{ct}^{(\mathsf{out})}\text{: a block that encrypts } m \in \{0,4,8,12\} \text{ if } \mathsf{msg} \geq 4 \\ \left(\mathsf{ct}^{(\mathsf{out},1)}, \mathsf{ct}^{(\mathsf{out},2)}\right)\text{: a pair of blocks that encrypt } m^{(i)} \in \{0,1\} \text{ otherwise} \end{cases}$

if msg ≥ 4 **then**
  Function `LocalFn`(x, y):
    /* Calls Algorithm 66 */
    $b_0 \leftarrow$ `IsXLessThanYGivenInputBorrow`$(x, y, 0)$
    $b_1 \leftarrow$ `IsXLessThanYGivenInputBorrow`$(x, y, 1)$
    return $((b_1 \ll 1) \mid b_0) \ll 2$
  return $\mathsf{BivPBS}\left(\mathtt{LocalFn}, \mathsf{ct}^{(\mathsf{lhs})}_{\mathcal{B}-1}, \mathsf{ct}^{(\mathsf{rhs})}_{\mathcal{B}-1}\right)$
else
  $b_0 \leftarrow \mathsf{BivPBS}\left((x,y) \mapsto \mathtt{IsXLessThanYGivenInputBorrow}(x,y,0), \mathsf{ct}^{(\mathsf{lhs})}_{\mathcal{B}-1}, \mathsf{ct}^{(\mathsf{rhs})}_{\mathcal{B}-1}\right)$
  $b_1 \leftarrow \mathsf{BivPBS}\left((x,y) \mapsto \mathtt{IsXLessThanYGivenInputBorrow}(x,y,1), \mathsf{ct}^{(\mathsf{lhs})}_{\mathcal{B}-1}, \mathsf{ct}^{(\mathsf{rhs})}_{\mathcal{B}-1}\right)$
  return (b_0, b_1)

Algorithm 68: Comparison: auxiliary function to finalize comparison

```
1  Function FinishCompare(groupBorrows, preparedSignedCheck, useSequentialAlgorithm, invertResult):
       Input:  groupBorrows = (ct_i^(groupBorrow)) : a tuple of group borrows
               preparedSignedCheck : result of PrepareSignedCheck for signed numbers, otherwise None
               useSequentialAlgorithm ∈ {true, false} : flag if the sequential algorithm is to be used
               invertResult ∈ {true, false} : flag if the result is to be inverted
       Output: ct^(result) : encrypted boolean
2      lastGroupBorrowState ← groupBorrows.pop()
3      if groupBorrows.isEmpty() then
4          return lastGroupBorrowState
5      else if useSequentialAlgorithm then
           /* Algorithm 33                                                          */
6          resolvedBorrows ← ResolveGroupCarriesSequentially(groupBorrows)
7      else
           /* Algorithm 35                                                          */
8          resolvedBorrows ← ResolveGroupsCarriesUsingHillisSteele(groupBorrows)
9      if preparedSignedCheck = None then
10         result ← lastGroupBorrowState + resolvedBorrows.last()
11         result ← PBS((x) ↦ ((x ≫ 1) & 1) ⊕ invertResult, result)
12         return result
13     else
14         if len(preparedSignedCheck) = 1 then
15             if len(resolvedBorrows) > 0 then
16                 lastGroupBorrowState ← lastGroupBorrowState + resolvedBorrows.last()
17             result ← lastGroupBorrowState + preparedSignedCheck
18             Function LocalFn(block):
19                 index ← if len(resolvedBorrows) > 0 then 0 else 1
20                 inputBorrw ← (block ≫ index) & 1
21                 if inputBorrow = 1 then
22                     return (block ≫ 3) & 1
23                 else
24                     return (block ≫ 2) & 1
25             result ← PBS(LocalFn, result)
26             return result
27         else
28             if len(resolvedBorrows) > 0 then
29                 lastGroupBorrowState ← lastGroupBorrowState + resolvedBorrows.last()
30                 lastGroupBorrowState ← PBS((x) ↦ x ≫ 1 & 1, lastGroupBorrowState)
31             result ← if lastGroupBorrowState then resolvedBorrows[1] else resolvedBorrows[0]
32             if invertResult then
33                 return BitwiseNOT(result)
34             else
35                 return result
```

Algorithm 69: $\mathsf{ct}^{(\mathsf{Compare})} \leftarrow \mathsf{Compare}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{lhs})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{rhs})}, \mathsf{kind}\right)$

Input: $\begin{cases} \overrightarrow{\mathsf{ct}}^{(\mathsf{lhs})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{rhs})} : \text{homomorphic integers} \\ \mathsf{kind} \in \{\mathsf{Less}, \mathsf{LessOrEqual}, \mathsf{Greater}, \mathsf{GreaterOrEqual}\} \end{cases}$

Output: $\mathsf{ct}^{(\mathsf{Compare})}$: encrypted boolean

switch kind **do**
  case Less **do**
    lhs, rhs, invertResult $\leftarrow \left(\overrightarrow{\mathsf{ct}}^{(\mathsf{lhs})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{rhs})}, \mathsf{false}\right)$
  case LessOrEqual **do**
    lhs, rhs, invertResult $\leftarrow \left(\overrightarrow{\mathsf{ct}}^{(\mathsf{rhs})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{lhs})}, \mathsf{true}\right)$
  case Greater **do**
    lhs, rhs, invertResult $\leftarrow \left(\overrightarrow{\mathsf{ct}}^{(\mathsf{rhs})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{lhs})}, \mathsf{false}\right)$
  case GreaterOrEqual **do**
    lhs, rhs, invertResult $\leftarrow \left(\overrightarrow{\mathsf{ct}}^{(\mathsf{lhs})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{rhs})}, \mathsf{true}\right)$

subBlocks $\leftarrow \{\}$
for $i \in [0, \mathcal{B}-1]$ **do**
  subBlocks.append($\mathsf{lhs}_i - \mathsf{rhs}_i$)

groupSize $\leftarrow \lfloor \log_2(\mathsf{p}) \rfloor$
firstGroupLuts, otherGroupLuts $\leftarrow$ `CreateBlockStatesLuts`(groupSize) ; // Algorithm 29
blocksStates $\leftarrow \{\}$
for $i \in [0, \mathcal{B} - 1 - \mathsf{isSigned}(\mathsf{lhs})]$ **do**
  groupIndex, indexInGroup $\leftarrow (i \;//\; \mathsf{groupSize}, i \bmod \mathsf{groupSize})$
  if groupIndex $= 0$ **then**
    blocksStates.append(PBS(firstGroupLuts[indexInGroup], $\mathsf{subBlocks}_\mathsf{i}$))
  else
    blocksStates.append(PBS(otherGroupLuts[indexInGroup], $\mathsf{subBlocks}_\mathsf{i}$))

groupBorrows $\leftarrow$ `ComputeGroupBorrowState`(invertResult, groupSize, blocksStates) ; // Algorithm 65
maybePreparedSignedCheck $\leftarrow$ None
if isSigned(lhs) **then**
  maybePreparedSignedCheck $\leftarrow$ `PrepareSignedCheck`($\mathsf{lhs}_{\mathcal{B}-1}, \mathsf{rhs}_{\mathcal{B}-1}$) ; // Algorithm 67
/* Final call to Algorithm 68 */
return `FinishCompare`(groupBorrows, maybePreparedSignedCheck, useSequentialAlgorithm, invertResult)

3.17 Division and Modulo

The division and modulo operations are the most challenging of the basic arithmetic operations.

3.17.1 Ciphertext-Ciphertext Division & Modulo

Unsigned. Division by an unsigned encrypted value is based on the binary long division algorithm, adapted to the homomorphic setting. Recall that for binary digits, when computing the digits of the quotient in this method, the next digit can be computed as the Boolean $r \geq d$ where d is the divisor and r is the value of the current remainder; then we branch on the value of this digit.

This branching typically requires the evaluation of an `if` statement, and, although conditional execution is not possible, the `if` statement can be replaced with a `select` (Section 3.15). However, this means the computations inside the `if` have to be done anyway. Thus, to make the algorithm FHE friendly, the comparison can be removed by computing the overflow of the subtraction $r - d$ since $r - d$ overflows if and only if $d > r$ which is if and only if $\neg(r \geq d)$.

Another improvement that can be made is splitting the remainder r and divisor d in two. As we know, the remainder starts with all bits set to 0 and at each iteration, one more bit is set, we know which part of r consists of 0 bits, allowing us to take the same part of d compare it with 0 and do a proper `overflowing_subtraction` for the other part and later combine the two together. This is something that is worth doing as it reduces the depth of the carry propagation needed during each iteration. This brings us to the division algorithm implemented inside TFHE-rs, which is presented in Algorithm 71 and uses Algorithm 70 as a subroutine.

Algorithm 70: Auxiliary function for the division

Function `SplitAt`$(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}, \mathsf{bitIndex})$:

Input:
- $\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$: a homomorphic integer
- $\mathsf{bitIndex} \in [0, \log_2(\Omega) - 1]$: bit index at which $\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$ shall be split.
- The bit at index $\mathsf{bitIndex}$ is contained in the second part

Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{low})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{high})}$:
two homomorphic integers encrypting the lower and higher order bits of the input

$\mathsf{blockIndex} \leftarrow \mathsf{bitIndex} \,//\, \log_2(\mathsf{msg})$

if $(\mathsf{bitIndex} \bmod \log_2(\mathsf{msg})) = 0$ **then**

  return $\left\{\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}_0, \cdots \overrightarrow{\mathsf{ct}}^{(\mathsf{in})}_{\mathsf{blockIndex}-1}\right\}, \left\{\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}_{\mathsf{blockIndex}}, \cdots, \overrightarrow{\mathsf{ct}}^{(\mathsf{in})}_{\mathcal{B}-1}\right\}$

else

  $\overrightarrow{\mathsf{ct}}^{(\mathsf{low})} \leftarrow \left\{\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}_0, \cdots \overrightarrow{\mathsf{ct}}^{(\mathsf{in})}_{\mathsf{blockIndex}-1}\right\}$

  $\overrightarrow{\mathsf{ct}}^{(\mathsf{low})}_{\mathsf{blockIndex}} \leftarrow \mathsf{PBS}\Big((x) \mapsto x \bmod (1 \ll (\mathsf{bitIndex} \bmod \log_2(\mathsf{msg}))), \overrightarrow{\mathsf{ct}}^{(\mathsf{in})}_{\mathsf{blockIndex}}\Big)$

  $\overrightarrow{\mathsf{ct}}^{(\mathsf{high})} \leftarrow \mathtt{RightShift}\Big(\left\{\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}_{\mathsf{blockIndex}}, \cdots, \overrightarrow{\mathsf{ct}}^{(\mathsf{in})}_{\mathcal{B}-1}\right\}, \mathsf{bitIndex} \bmod \log_2(\mathsf{msg})\Big)$

  return $\overrightarrow{\mathsf{ct}}^{(\mathsf{low})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{high})}$

Algorithm 71: $\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{q})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{r})}\right) \leftarrow \mathsf{UnsignedDivRem}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{n})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{d})}\right)$

Input: $\begin{cases}\overrightarrow{\mathsf{ct}}^{(\mathsf{n})}\text{: a homomorphic integer encrypting the numerator of the division}\\ \overrightarrow{\mathsf{ct}}^{(\mathsf{d})}\text{: a homomorphic integer encrypting the denominator of the division}\end{cases}$

Output: $\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{q})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{r})}\right)$: two homomorphic integers encrypting $m^{(\mathsf{q})}$, $m^{(\mathsf{r})}$ where $m^{(\mathsf{n})} = m^{(\mathsf{q})} \cdot m^{(\mathsf{d})} + m^{(\mathsf{r})}$ with $0 \leq m^{(\mathsf{r})} < m^{(\mathsf{d})}$

$\overrightarrow{\mathsf{ct}}^{(\mathsf{r})} \leftarrow \left\{\mathsf{ct}^{(\mathsf{Triv})}(0)\right\}_{i\in[0,\mathcal{B}-1]}$

$d_{\text{len}} \leftarrow \mathsf{numBitsOf}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{d})}\right)$

for $i \in [0, d_{\text{len}}]$ **do**

  $\overrightarrow{\mathsf{ct}}^{(\mathsf{r})} \leftarrow \mathsf{LeftShift}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{r})}, 1\right)$

  /* Set the LSB of r equal to the i-th bit of n */

  $\mathsf{ct}_0^{(\mathsf{r})} \leftarrow \mathsf{ct}_0^{(\mathsf{r})} + \mathsf{PBS}\left(i\text{-th bit of } x, \overrightarrow{\mathsf{ct}}^{(\mathsf{n})}\right)$

  $\overrightarrow{\mathsf{ct}}^{(\mathsf{r})}_{\mathsf{high}}, \overrightarrow{\mathsf{ct}}^{(\mathsf{r})}_{\mathsf{low}} \leftarrow \mathtt{SplitAt}(\overrightarrow{\mathsf{ct}}^{(\mathsf{r})}, i)$

  $\overrightarrow{\mathsf{ct}}^{(\mathsf{d})}_{\mathsf{high}}, \overrightarrow{\mathsf{ct}}^{(\mathsf{d})}_{\mathsf{low}} \leftarrow \mathtt{SplitAt}(\overrightarrow{\mathsf{ct}}^{(\mathsf{d})}, i)$

  $\overrightarrow{\mathsf{ct}}^{(\mathsf{r.new})}, \mathsf{ct}^{(\mathsf{overflow})} \leftarrow \mathsf{overflowingSub}(\overrightarrow{\mathsf{ct}}^{(\mathsf{r})}_{\mathsf{low}}, \overrightarrow{\mathsf{ct}}^{(\mathsf{d})}_{\mathsf{low}})$

  $\mathsf{ct}^{(\mathsf{overflow})} \leftarrow \mathsf{Bitwise}\left(\mathsf{OR}, \mathsf{ct}^{(\mathsf{overflow})}, \mathsf{Neq}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{d})}_{\mathsf{high}}, 0\right)\right)$

  $\overrightarrow{\mathsf{ct}}^{(\mathsf{r})} \leftarrow \mathsf{Select}\left(\mathsf{ct}^{(\mathsf{overflow})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{r})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{r.new})}\right)$

  /* Set the i-th bit of q; simplified */

  $\mathsf{ct}_i^{(\mathsf{q})} \leftarrow \mathsf{Neg}\left(\mathsf{ct}^{(\mathsf{overflow})}\right)$

return $\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{q})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{r})}\right)$

Theorem 30 *Let* $\overrightarrow{\mathsf{ct}}^{(\mathsf{n})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{d})}$ *be two large homomorphic integers encrypting unsigned values* $m^{(n)}$ *and* $m^{(d)}$*, respectively. Then,*

$$\mathsf{UnsignedDivRem}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{n})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{d})}\right) = \left(\overrightarrow{\mathsf{ct}}^{(\mathsf{q})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{r})}\right) \textit{ with}$$

$$\mathsf{Dec_s}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{q})}\right) = m^{(\mathsf{n})} \mathbin{//} m^{(\mathsf{d})} \quad \textit{and} \quad \mathsf{Dec_s}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{r})}\right) = m^{(\mathsf{n})} \bmod m^{(\mathsf{d})}.$$

Signed. The signed integer division relies on the unsigned division, with a few pre-processing and post-processing steps. The algorithm used for signed integers gives a division that rounds towards 0.

Algorithm 72: $\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{q})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{r})}\right) \leftarrow \mathsf{SignedDivRem}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{n})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{d})}\right)$

Input: $\begin{cases}\overrightarrow{\mathsf{ct}}^{(\mathsf{n})}\text{: a homomorphic integer encrypting the numerator of the division}\\ \overrightarrow{\mathsf{ct}}^{(\mathsf{d})}\text{: a homomorphic integer encrypting the denominator of the division}\end{cases}$

Output: $\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{q})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{r})}\right)$: two homomorphic integers encrypting $m^{(\mathsf{q})},\ m^{(\mathsf{r})}$ where $m^{(\mathsf{n})} = m^{(\mathsf{q})} \cdot m^{(\mathsf{d})} + m^{(\mathsf{r})}$ with $-m^{(\mathsf{d})} < m^{(\mathsf{r})} < m^{(\mathsf{d})}$

$\overrightarrow{\mathsf{ct}}^{(\mathsf{d}_+)} \leftarrow \mathsf{Abs}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{d})}\right)$

$\overrightarrow{\mathsf{ct}}^{(\mathsf{n}_+)} \leftarrow \mathsf{Abs}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{n})}\right)$

/* position within the most significant block */

$\mathsf{signBitPos} \leftarrow \log_2(\mathsf{msg}) - 1$

$\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{q})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{r})}\right) \leftarrow \mathsf{UnsignedDivRem}\left(\overrightarrow{\mathsf{ct}}^{(\mathsf{n}_+)}, \overrightarrow{\mathsf{ct}}^{(\mathsf{d}_+)}\right)$

$\mathsf{lut} \leftarrow \mathbf{GenLUT}\left((x, y) \mapsto (x \gg \mathsf{signBitPos}) \neq (y \gg \mathsf{signBitPos})\right)$

$\mathsf{signBitsAreDifferent} \leftarrow \mathsf{BivPBS}\left(\mathsf{lut}, \overrightarrow{\mathsf{ct}}^{(\mathsf{n})}_{\mathcal{B}-1}, \overrightarrow{\mathsf{ct}}^{(\mathsf{d})}_{\mathcal{B}-1}\right)$

$\overrightarrow{\mathsf{ct}}^{(q)} \leftarrow \mathsf{select}\left(\mathsf{signBitsAreDifferent}, -\overrightarrow{\mathsf{ct}}^{(\mathsf{q})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{q})}\right)$

$\mathsf{numeratorIsNegative} \leftarrow \mathsf{PBS}\left((x) \mapsto x \gg \mathsf{signBitPos}, \overrightarrow{\mathsf{ct}}^{(n)}_{\mathcal{B}-1}\right)$

$r \leftarrow \mathsf{select}\left(\mathsf{numeratorIsNegative}, -\overrightarrow{\mathsf{ct}}^{(\mathsf{r})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{r})}\right)$

return $\overrightarrow{\mathsf{ct}}^{(\mathsf{q})}, \overrightarrow{\mathsf{ct}}^{(\mathsf{r})}$

3.17.2 Plaintext-Ciphertext Division & Modulo

Dividing by a clear value is significantly more efficient due to the large performance gap between clear operations and FHE operations. A common optimization used by compilers and big-integer libraries is to replace division with multiplication by the precomputed inverse.

Our division and modulo by known value are using the algorithm described in *Division by Invariant Integers using Multiplication* [GM94]. The computation of the inverse happens in the clear, and then the actual division/modulo uses operations already defined in this document.

4 LWE Ciphertext Compression

The *expansion factor* of an LWE ciphertext $\mathsf{ct} \in \mathsf{LWE}_{\mathbf{s}}(\widetilde{m}) \subseteq \mathbb{Z}_q^{n+1}$ is given by the ratio between the ciphertext size and the plaintext size. As an example, the default parameters in TFHE-rs are `V1_0_PARAM_MESSAGE_2_CARRY_2_KS_PBS_TUNIFORM_2M128` and use $n = 879$ and $q = 2^{64}$ meaning that the ciphertext size is $880 \cdot 64 = 56,320$ bits. In this parameter set, each ciphertext encrypts just two bits, meaning that the expansion factor is $56,320/2 = 28,160$. As we can see, LWE ciphertexts have a large expansion factor and it is therefore desirable to compress ciphertexts at various stages of FHE computation. There are three types of compression which are particularly useful:

1. Input ciphertext compression: this comprises of compressing LWE ciphertexts, typically generated by a client, which are to be sent to a server for homomorphic computation.
2. Intermediate ciphertext compression: this comprises of compressing LWE ciphertexts during computation (i.e., these ciphertexts are both outputs of previous computation and are required to be inputs to future computation). This compression occurs entirely on the server.
3. Output ciphertext compression: this comprises of compressing LWE ciphertexts which contain the result of a homomorphic computation, and are to be sent back to the client for decryption.

Note that input and output compression are typically aimed at saving *bandwidth*, whilst intermediate compression is aimed at saving *storage*. The input ciphertext compression technique in TFHE-rs uses a CSPRNG to compress the random mask terms to a seed, and is explained in Remark 5.

4.1 Input Ciphertext Compression

In Table 1, we present all relevant object sizes for the TFHE-rs scheme. In particular, we outline (theoretical) sizes of all secret keys, ciphertexts, and evaluation keys used as part of the TFHE scheme, including those for fresh ciphertexts compressed using a CSPRNG (see Remark 5).

Object	**Size (bits)**	
	Uncompressed	**CSPRNG Compressed**
$\mathbf{s}$	n	—
$\mathbf{S}$	kN	—
LWE ciphertext	$(n+1)\log_2 q$	$\rho + \log_2 q$
GLWE ciphertext	$(k+1)N\log_2 q$	$\rho + N\log_2 q$
GLev ciphertext	$\ell(k+1)N\log_2 q$	$\rho + \ell N\log_2 q$
GGSW ciphertext	$\ell(k+1)^2 N\log_2 q$	$\rho + \ell(k+1)N\log_2 q$
BSK	$n\ell_{\mathsf{BSK}}(k+1)^2 N\log_2 q$	$\rho + n\ell_{\mathsf{BSK}}(k+1)N\log_2 q$
MB-BSK	$2^{\mathsf{gf}} \cdot \frac{n}{\mathsf{gf}} \cdot \ell_{\mathsf{BSK}}(k+1)^2 N\log_2 q$	$\rho + 2^{\mathsf{gf}} \cdot \frac{n}{\mathsf{gf}} \cdot \ell_{\mathsf{BSK}}(k+1)N\log_2 q$
LWEKSK	$\ell_{\mathsf{KSK}}(n+1)kN\log_2 q$	$\rho + kN\ell_{\mathsf{KSK}}\log_2 q$
PKSK	$n\ell_{\mathsf{PKS}}(\bar{k}+1)\bar{N}\log_2 q$	$\rho + n\ell_{\mathsf{PKS}}\bar{N}\log_2(q)$
DMK	$Z(n+1)\log_2 q$	$\rho + Z\log_2 q$

Table 1: Sizes of ciphertexts and evaluation key material in TFHE, ρ denotes the bit size of the seed used in the CSPRNG.

4.2 Intermediate Ciphertext Compression

The most complex setting for compression is the intermediate case. Here, the ciphertexts to be compressed are assumed to be the output of a previous bootstrap (and therefore have a higher noise level than fresh LWE ciphertexts). These ciphertexts must be stored for future computation, under the constraint that we want the storage requirement to be as small as possible (i.e, the expansion factor should be minimized). The technique used within the TFHE-rs library for this is a combination of a packing-keyswitch and a modulus switch. In particular, given a homomorphic integer $\overrightarrow{\mathsf{ct}}$ which is made up of individual LWE ciphertexts encrypting the messages $m_0, m_1, \cdots m_{\mathcal{B}-1}$, the packing keyswitch packs these messages into a GLWE ciphertext encrypting the message $\sum_{j=0}^{\mathcal{B}-1} m_j \cdot X^{i_j}$ for a given set of indices $0 \leq i_0 < i_1 < \cdots < i_{\mathcal{B}-1} \leq N-1$.

$$\mathsf{CT}_{\mathsf{pack}} \leftarrow \mathsf{PackingKS}(\{\mathsf{ct}_j\}_{j=0}^{\mathcal{B}-1}, \{i_j\}_{j=0}^{\mathcal{B}-1}, \mathsf{KSK}) \in \mathsf{GLWE}\left(\sum_{j=0}^{\mathcal{B}-1} m_j \cdot X^{i_j}\right)$$

which generates a GLWE encryption of $\sum_{j=0}^{\mathcal{B}-1} m_j \cdot X^{i_j}$. This is followed by a modulus switch to the storage modulus $\bar{q}$ (note that TFHE-rs typically chooses $\bar{q} = 2N$):

$$\mathsf{CT}_{\mathsf{Compress}} \leftarrow \mathsf{MS}(\mathsf{ct}_{\mathsf{in}}, \bar{q})$$

to produce the output, compressed, GLWE ciphertext $\mathsf{CT}_{\mathsf{Compress}}$. Decompression is computed via a combination of sample extractions and blind rotations. This technique allows us to pack $\mathcal{B} \leq N$ LWE ciphertexts (the default parameters in the TFHE-rs library support compression of up to 256 individual LWE ciphertexts into a single GLWE ciphertext) of original size $\mathcal{B} \cdot (n+1) \cdot \log_2(q)$ into a single GLWE ciphertext of size $(\bar{k}+1) \cdot \bar{N} \cdot \log_2(\bar{q})$ where $(\bar{k}, \bar{N}, \bar{q})$ are compression-specific GLWE parameters. When using the right parameters, the expansion factor can be reduced from $28,160$ to 30 for 256 LWEs packed into a single GLWE ciphertext. We note that lower expansion factors can be achieved (either by packing a greater number of LWE ciphertexts, or via a different parameter optimization) however the associated cost can increase. The core flow of our compression algorithm can be seen in Fig. 8.

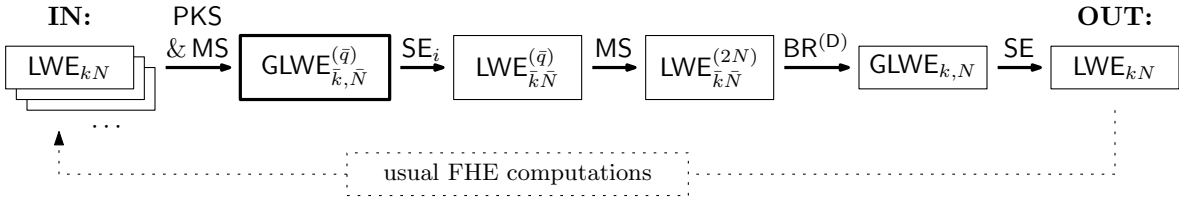

Figure 8: Computation flow for the intermediate compression: a list of α LWE's at the input, the compressed ciphertext given in a bold box, the i-th decompressed sample at the output. PKS+MS denotes the application of a packing keyswitch and modulus switch operation, SE denotes a sample extraction, MS denotes a modulus switch, $\mathsf{BR}^{(D)}$ denotes blind rotation with the usage of a decompression key. Ciphertexts' dimensions (and polynomial degrees) are given as a subscript, and the modulus is the default one (i.e., q) unless stated explicitly in the superscript. The dimensions (and polynomial degrees) are: $k \cdot N$ for the input LWE's, $(\bar{k}, \bar{N})$ for PKS. The modulus is denoted using the superscript, and the storage modulus $\bar{q}$ is typically set to be $2N$ in TFHE-rs.

We now define the compression and decompression algorithms. Here, we consider the packing and unpacking of a *single* homomorphic integer of generic size. We note that several homomorphic integers can be packed together, simply by viewing them as a single large integer. For a generic description of

the packing keyswitch algorithm, which is a major subroutine used in the packing, we refer the reader to Algorithm 13.

Algorithm 73: $\mathsf{CT}_{\mathsf{Compress}} \leftarrow \mathsf{Compress}(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}, \mathsf{KSK}, \mathcal{I}, q_{\mathsf{stg}})$

Input:
- $\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}$: a homomorphic integer
- KSK: a packing keyswitching key
- $\mathcal{I} = \{i_j\}_{j=0}^{\mathcal{B}-1}$: the indices used to pack the messages, with $0 \leq i_0 < i_1 < \cdots < i_{\mathcal{B}-1} < N$
- q_{stg}: the storage modulus

Output: $\mathsf{CT}^{(\mathsf{out})} \in \mathsf{GLWE}_{\mathbf{S}}\left(\sum_{j=0}^{\mathcal{B}-1} \widetilde{m_j} \cdot X^{i_j}\right) \subseteq \mathfrak{R}_{q_{\mathsf{stg}}}^{(k+1)}$

$\mathsf{CT}^{(\mathsf{pack})} \leftarrow \mathsf{PackingKS}(\overrightarrow{\mathsf{ct}}^{(\mathsf{in})}, \mathcal{I}, \mathsf{KSK})$

$\mathsf{CT}^{(\mathsf{Compress})} \leftarrow \mathsf{MS}\left(\mathsf{CT}^{(\mathsf{pack})}, q_{\mathsf{stg}}\right)$

return $\mathsf{CT}^{(\mathsf{Compress})}$

Algorithm 74: $\overrightarrow{\mathsf{ct}}^{(\mathsf{Decompress})} \leftarrow \mathsf{Decompress}(\mathsf{CT}, \mathsf{BSK}, \mathcal{I})$

Input:
- $\mathsf{CT}^{(\mathsf{in})}$: a compressed GLWE ciphertext
- BSK: a decompression (bootstrapping) key
- $\mathcal{I} = \{i_j\}_{j=0}^{\mathcal{B}-1}$: the indices of the packed messages, with $0 \leq i_0 < i_1 < \cdots < i_{\mathcal{B}-1} < N$

Output: $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$: a homomorphic integer

for $0 \leq j \leq \mathcal{B}-1$ **do**

  $\mathsf{ct}^{(\mathsf{ext})} \leftarrow \mathsf{SE}\left(\mathsf{CT}^{(\mathsf{in})}, i_j\right)$

  $\mathsf{ct}^{(\mathsf{ms})} \leftarrow \mathsf{MS}\left(\mathsf{ct}^{(\mathsf{ext})}, 2N\right)$

  $\mathsf{CT}^{(\mathsf{BRot})} \leftarrow \mathsf{BlindRotate}\left(\mathsf{ct}^{(\mathsf{ms})}, \mathsf{BSK}\right)$

  $\mathsf{ct}_j^{(\mathsf{out})} \leftarrow \mathsf{SE}\left(\mathsf{CT}^{(\mathsf{BRot})}, 0\right)$

return $\overrightarrow{\mathsf{ct}}^{(\mathsf{out})}$

Theorem 31 (LWE Ciphertext Compression) *Let $\overrightarrow{\mathsf{ct}}$ be a homomorphic integer containing $\mathcal{B}$ blocks, with $\overrightarrow{\mathsf{ct}}_i = \mathsf{LWE}_{\mathbf{s}}(m_i)$, $0 \leq i \leq \mathcal{B}-1$, let $\mathcal{I} = \{i_j\}_{j=0}^{\mathcal{B}-1}$ be the set of indices for the packed messages, with $0 \leq i_0 < i_1 < \cdots < i_{\mathcal{B}-1} \leq N-1$. Then:*

$$\mathsf{Compress}(\overrightarrow{\mathsf{ct}}) = \mathsf{GLWE}_{\mathbf{S}}\left(\sum_{j=0}^{\mathcal{B}-1} m_j \cdot X^{i_j}\right)$$

The algorithmic complexity is: $\mathbb{C}^{\mathsf{Compress}} = \mathbb{C}^{\mathsf{PKS}} + \mathbb{C}^{\mathsf{MS}}$

Theorem 32 (LWE Ciphertext Decompression) *Let $\mathsf{CT} = \mathsf{GLWE}_{\mathbf{S}}\left(\sum_{j=0}^{\mathcal{B}-1} m_j \cdot X^{i_j}\right)$ be a compressed ciphertext containing $\mathcal{B}$ LWE ciphertexts, $\mathcal{I} = \{i_j\}_{j=0}^{\mathcal{B}-1}$ be the set of indices for the packed messages, with $0 \leq i_0 < i_1 < \cdots < i_{\mathcal{B}-1} \leq N-1$. Then:*

$$\mathsf{Decompress}(\mathsf{CT}) = \{\mathsf{LWE}_{\mathbf{s}}(\widetilde{m_j})\}_{j=0}^{\mathcal{B}-1}$$

The algorithmic complexity is: $\mathbb{C}^{\mathsf{Decompress}} = \mathbb{C}^{\mathsf{SE}} + \mathbb{C}^{\mathsf{MS}} + \mathbb{C}^{\mathsf{BR}} + \mathbb{C}^{\mathsf{SE}}$

Remark 38 (Output Ciphertext Compression) *TFHE-rs uses the same technique as the intermediate ciphertext compression method for output ciphertext compression.*

5 Benchmarks and Parameters

In this section, we will outline the different parameters used inside TFHE-rs and their associated benchmarks with the various available operations. We are only giving benchmarks assuming the usual Gaussian distribution to simplify comparison with other work in the literature. Note that in TFHE-rs, the default distribution is TUniform to better fit some use-cases. TUniform benchmarks can be found online and we refer to https://docs.zama.ai/tfhe-rs/get-started/benchmarks for these benchmarks. As a general rule of thumb, switching to a TUniform distribution for the error leads to timings which are roughly between 5 to 10 percent slower. All benchmarks presented in this section are CPU-based, and were performed on AWS using a `hpc7a.96xlarge`[10] machine.

5.1 Bootstrapping Related Benchmarks

We begin by looking at benchmarks for the PBS operation, which is the atomic unit of circuit cost when evaluating algorithms using the TFHE scheme. In particular, we consider benchmarks of the *classical bootstrapping* in Section 5.1.1 (cf. Algorithm 19), as well as the *multi-bit bootstrapping* in Section 5.1.2 (cf. Algorithm 20).

5.1.1 Classical Bootstrapping Performance

The first benchmark of interest, presented in Table 2, is to consider the cost of the programmable bootstrapping operation for a variety of precisions $\log_2(p)$ (we consider 2, 4, 6, and 8 in this section). In particular we measure the cost of the *keyswitch* (KS), the *programmable bootstrap* (PBS) as well as combination of the two (KS-PBS). For the KS-PBS operation, we consider three separate timings: the latency of the KS-PBS operation, an amortized "per-bit" cost of the KS-PBS operation, and the latency of the KS-PBS operation when combined with the drift mitigation techniques outlined in Algorithm 15.

A key observation in Table 2 is that, for most failure probabilities, the amortized cost of the KS-PBS operation—defined as the KS-PBS latency divided by the number of bits bootstrapped—is optimal when choosing $\log_2(p) = 4$. This makes $\mathsf{p} = 2^4$ a strong candidate for constructing homomorphic integers, whose benchmarks are discussed in Section 5.2.

[10] https://aws.amazon.com/ec2/instance-types/hpc7a/

p_{fail}	Operation	$\log_2(p)$			
		2	4	6	8
2^{-40}	KS	677 μs	2.29 ms	20.9 ms	53.3 ms
2^{-40}	PBS	5.77 ms	10.7 ms	94.2 ms	471 ms
2^{-40}	KS-PBS	7.01 ms	13.6 ms	109 ms	513 ms
2^{-40}	KS-PBS (amortized)	7.01 ms	**6.80 ms**	36.33 ms	128.25 ms
2^{-64}	KS	1.28 ms	2.39 ms	15.4 ms	130 ms
2^{-64}	PBS	8.33 ms	11.2 ms	99.2 ms	641 ms
2^{-64}	KS-PBS	9.74 ms	14.2 ms	116 ms	791 ms
2^{-64}	KS-PBS (amortized)	10.2 ms	**7.14 ms**	38.67 ms	158.5 ms
2^{-128}	KS	1.16 ms	5.28 ms	30.3 ms	154 ms
2^{-128}	PBS	8.67 ms	25.2 ms	212 ms	1.26 s
2^{-128}	KS-PBS	10.4 ms	31.1 ms	238 ms	1.58 s
2^{-128}	KS-PBS (amortized)	**10.4 ms**	12.4 ms	47.33 ms	316.0 ms
2^{-128}	KS-PBS (drift)	10.4 ms	14.6 ms	122 ms	1.41s
2^{-128}	KS-PBS (drift, amortized)	10.4 ms	**7.3 ms**	20.3 ms	176.3 ms

Table 2: Benchmarks for a variety of atomic patterns for different failure probabilities with parameters using a Gaussian distribution. The parameter set in each case corresponds to X_X_Y in Table 13, where $X = \log_2(p)/2$ is the precision used and $Y = \log_2(p_{\mathsf{fail}})$. The timings labelled "KS-PBS (amortized)" denote the timing of the KS-PBS divided by the number of bits in the corresponding operation, i.e., it represents $\frac{1}{\log_2(p)} \cdot$ KS-PBS. "KS-PBS (drift)" denotes the latency of the KS-PBS operation when evaluating the KS-PBS operation in combination with the drift mitigation technique outlined in Algorithm 15.

5.1.2 MultiBit Bootstrapping Performance

In this section, we focus on the timings of parallelized variant of bootstrapping called the Multi-Bit PBS, described in Algorithm 20. Unlike in the usual bootstrapping, another parameter called the *grouping factor* (gf) has to be taken into account in these benchmarks. In Table 3, we show the timings depending on the input plaintext precision given as a bit size ($\log_2(p)$) and the grouping factor (gf). We note that in Table 3, for $p_{\mathsf{fail}} < 2^{-128}$, the drift mitigation technique is not used in any of the benchmarks.

p_{fail}	Operation	$\log_2(p) = 2$			$\log_2(p) = 4$			$\log_2(p) = 6$			$\log_2(p) = 8$		
		gf = 2	gf = 3	gf = 4	gf = 2	gf = 3	gf = 4	gf = 2	gf = 3	gf = 4	gf = 2	gf = 3	gf = 4
2^{-40}	KS	**669 us**	1.16 ms	1.2 ms	**1.23 ms**	1.31 ms	1.34 ms	15.5 ms	**14.1 ms**	15.1 ms	52.9 ms	**51.1 ms**	76.6 ms
2^{-40}	MB-PBS	5.91 ms	3.27 ms	**3.19 ms**	8.16 ms	5.05 ms	**4.06 ms**	51.3 ms	35.2 ms	**25.5 ms**	363 ms	**205 ms**	156 ms
2^{-40}	KS-MB-PBS	5.74 ms	**4.54 ms**	5.01 ms	8.55 ms	6.46 ms	**5.83 ms**	63.7 ms	**47.1 ms**	54.1 ms	442 ms	270 ms	**258 ms**
2^{-64}	KS	**1.07 ms**	1.22 ms	1.52 ms	1.8 ms	**1.39 ms**	1.93 ms	15.7 ms	**15.4 ms**	15.5 ms	130 ms	**125 ms**	130 ms
2^{-64}	MB-PBS	7.98 ms	4.37 ms	**3.23 ms**	8.83 ms	6.46 ms	**4.1 ms**	49.7 ms	41.4 ms	**26.8 ms**	395 ms	289 ms	**216 ms**
2^{-64}	KS-MB-PBS	8.19 ms	**5.17 ms**	5.51 ms	10.7 ms	7.37 ms	**6.51 ms**	66.8 ms	53.8 ms	**44.9 ms**	533 ms	396 ms	**389 ms**
2^{-80}	KS	1.25 ms	**1.14 ms**	1.52 ms	4.28 ms	**4.24 ms**	4.33 ms	**29.9 ms**	31.0 ms	30.8 ms	191 ms	**156 ms**	286 ms
2^{-80}	MB-PBS	5.96 ms	4.89 ms	**3.27 ms**	11.7 ms	7.68 ms	**5.82 ms**	101 ms	68.6 ms	**56.0 ms**	785 ms	515 ms	**412 ms**
2^{-80}	KS-MB-PBS	7.24 ms	**5.36 ms**	7.06 ms	16.2 ms	15.0 ms	**12.0 ms**	125 ms	97.5 ms	**87.6 ms**	834 ms	**616 ms**	637 ms
2^{-128}	KS	1.21 ms	**1.2 ms**	1.78 ms	4.06 ms	**3.94 ms**	4.24 ms	30.6 ms	**30.5 ms**	31.2 ms	167 ms	**148 ms**	159 ms
2^{-128}	MB-PBS	8.39 ms	5.17 ms	**4.53 ms**	14.7 ms	11.9 ms	**9.26 ms**	103 ms	81.2 ms	**62.6 ms**	845 ms	635 ms	**525 ms**
2^{-128}	KS-MB-PBS	7.26 ms	**6.09 ms**	6.39 ms	18.9 ms	16.0 ms	**13.5 ms**	131 ms	104 ms	**94.3 ms**	1.07 s	859 ms	**606 ms**

Table 3: Performance comparison of different operations grouped by precision across different grouping factors $\mathsf{gf} = 2$, $\mathsf{gf} = 3$, and $\mathsf{gf} = 4$ considered in the MB-PBS. Bold values indicate the smallest timing for each operation per $\log_2(p)$ column. All parameter sets used are outlined in Table 14

.

Recall that in the case of $\mathsf{gf} > 1$, we see an increase in the bootstrapping key size, as can be seen in Table 1. Table 3 shows that, as the precision increases, the KS-MB-PBS operation is consistently fastest in the case of $\mathsf{gf} = 4$. However, in this case, the bootstrapping keys are significantly larger (by a factor of $\frac{2^{\mathsf{gf}}}{\mathsf{gf}}$) which can be cumbersome to deal with when deploying advanced FHE protocols, particularly including those utilising threshold decryption, where keysize should be minimized to benefit the associated MPC protocols.

5.2 Integer Benchmarks

In this section, we consider the benchmark of homomorphic integer operations. In particular, the algorithms presented in Section 3. In Section 5.2.1, we discuss how to choose the best carry-message modulus to build large integers. Next, in Section 5.2.2, we consider the benchmark of plaintext-ciphertext operations (where algorithms have one encrypted input and one plaintext input) as well as ciphertext-ciphertext operations (where algorithms have only encrypted inputs) in Section 5.2.3. We also look at benchmark timings for the compression and decompression algorithms (Algorithms 73 and 74 respectively). Finally, we present all parameter sets used in Section 5.4.

5.2.1 Choosing the Best Precision and Bootstrapping Algorithm

The first step in constructing efficient homomorphic integers (and circuits for evaluating them) is selecting an appropriate carry-message modulus p for their representation. Suppose we want to evaluate operations on encrypted u64 values. In that case, we must determine the most efficient way to split this integer into smaller blocks for evaluation. As an example, we could split the 64-bit values into 32 blocks of 2-bits, or 16 blocks of 4-bits, etc. The message modulus msg directly impacts the required PBS precision, and therefore the latency of the individual bootstraps. Whilst choosing a lower precision bootstrap can lead to significantly faster PBS evaluation, it typically also leads to a circuit containing substantially more PBS operations, and can therefore lead to a negative trade-off.

To illustrate this with an example, Table 4 shows that the number of PBS operations required for a 64-bit multiplication circuit can range from as low as 341 (when using a message modulus $\mathsf{msg} = 2^4$ and a carry modulus $\mathsf{carry} = 2^4$, i.e., a carry-message modulus $\mathsf{p} = 2^8$) to as high as 6068 (when using a message modulus $\mathsf{msg} = 2^1$). However, the optimal choice is a message modulus $\mathsf{msg} = 2^2$, as indicated by the latency results for multiplication in Table 5.

Parameters	$\log_2(\Omega)$			
	8	16	32	64
1_1_64	92	372	1504	6068
2_2_64	25	116	455	1772
3_3_64	13	59	184	687
4_4_64	6	22	94	341

Table 4: The number of programmable bootstrapping operations in a $\log_2(\Omega)$-bit multiplication circuit, for a variety of PBS precisions. In a parameter set X_Y_Z, X refers to the $\log_2$ of the carry modulus, Y refers to the $\log_2$ of the message modulus and Z refers to the $\log_2$ of the failure probability.

Parameters	$\log_2(\Omega)$						
	4	8	16	32	64	128	256
Bitwise AND							
1_1_64	**14.9 ms**	**15.3 ms**	**16.1 ms**	**17.3 ms**	**18.8 ms**	**19.7 ms**	34.2 ms
2_2_64	18.9 ms	18.6 ms	19.0 ms	19.4 ms	20.3 ms	22.0 ms	**24.6 ms**
4_4_64	762 ms	844 ms	920 ms	1.03 s	1.09 s	1.12 s	1.15 s
Addition							
1_1_64	46.6 ms	124 ms	178 ms	370 ms	898 ms	1.43 s	2.85 s
2_2_64	**32.1 ms**	**52.5 ms**	**58.1 ms**	**79.9 ms**	**101 ms**	**161 ms**	**173 ms**
4_4_64	798 ms	1.64 s	2.78 s	3.09 s	3.23 s	4.17 s	4.8 s
Multiplication							
1_1_64	67.2 ms	164 ms	331 ms	653 ms	1.46 s	3.98 s	12.3 s
2_2_64	**38.5 ms**	**94.4 ms**	**136 ms**	**210 ms**	**381 ms**	**1.1 s**	**3.65 s**
4_4_64	745 ms	1.65 s	3.68 s	5.01 s	7.49 s	17.1 s	40.9 s

Table 5: Comparison of Bitwise AND, Addition, and Multiplication timings depending on the input size chunks and number of chunks. All parameter sets used have a failure probability satisfying $p_{\mathsf{fail}} \leq 2^{-64}$.

Parameters	$\log_2(\Omega)$						
	4	8	16	32	64	128	256
Bitwise AND							
`1_1_128`	**15.0 ms**	**15.7 ms**	**16.6 ms**	**18.0 ms**	**19.1 ms**	**20.9 ms**	37.3 ms
`2_2_128`	22.8 ms	19.7 ms	20.2 ms	20.9 ms	22.3 ms	24.7 ms	**27.3 ms**
`4_4_128`	1.47 s	1.63 s	1.73 s	1.78 s	1.89 s	2.04 s	2.08 s
Addition							
`1_1_128`	61.6 ms	123 ms	249 ms	517 ms	1.02 s	2.03 s	3.96 s
`2_2_128`	**42.6 ms**	**62.8 ms**	**59.2 ms**	**85.5 ms**	**105 ms**	**174 ms**	**190 ms**
`4_4_128`	1.51 s	3.22 s	5.1 s	5.13 s	5.76 s	8.12 s	8.34 s
Multiplication							
`1_1_128`	76.2 ms	176 ms	347 ms	723 ms	1.67 s	4.54 s	13.7 s
`2_2_128`	**41.5 ms**	**101 ms**	**146 ms**	**219 ms**	**400 ms**	**1.13 s**	**3.77 s**
`4_4_128`	1.39 s	3.25 s	6.82 s	9.14 s	19.0 s	52.4 s	184 s

Table 6: Comparison of Bitwise AND, Addition, and Multiplication timings depending on the input size chunks and number of chunks. All parameter sets used have a failure probability satisfying $p_{\mathsf{fail}} \leq 2^{-128}$.

We also observe that addition is most efficient when using a message modulus of $\mathsf{msg} = 2^2$ and a carry modulus of $\mathsf{carry} = 2^2$. This is also the best representation for the bitwise AND operator when handling large precisions (e.g., more than 256 in this case). These findings suggest that a 4-bit carry-message modulus is a good starting point.

In Table 7, we compare classical bootstrapping with multi-bit bootstrapping using a grouping factor of 3, both for $\mathsf{p} = 2^4$. While multi-bit PBS is faster for small precisions, it quickly becomes slower than the classical approach for larger precisions (i.e., when $\log_2(\Omega) \geq 64$).

For this reason, the default representation of homomorphic 64-bit integers in TFHE-rs uses a carry modulus of $\mathsf{carry} = 2^2$ and a message modulus of $\mathsf{msg} = 2^2$, relying on classical PBS rather than later variants like multi-bit PBS.

Parameters	$\log_2(\Omega)$						
	4	8	16	32	64	128	256
Bitwise AND							
`2_2_64`	18.9 ms	18.6 ms	19.0 ms	19.4 ms	**20.3 ms**	**22.0 ms**	**24.6 ms**
`2_2_64_MultBit3`	**13.8 ms**	**12.6 ms**	**13.6 ms**	**17.3 ms**	26.8 ms	45.1 ms	93.6 ms
Addition							
`2_2_64`	32.1 ms	52.5 ms	58.1 ms	79.9 ms	**101 ms**	**161 ms**	**173 ms**
`2_2_64_MultBit3`	**24.1 ms**	**37.1 ms**	**40.6 ms**	**64.8 ms**	106 ms	201 ms	395 ms
Multiplication							
`2_2_64`	38.5 ms	94.4 ms	136 ms	**210 ms**	**381 ms**	**1.1 s**	**3.65 s**
`2_2_64_MultBit3`	**25.3 ms**	**69.9 ms**	**133 ms**	359 ms	1.15 s	4.14 s	15.8 s

Table 7: Comparison of Bitwise AND, Addition, and Multiplication timings depending on the bootstrapping method and number of chunks. Both parameter sets are chosen so that $p_{\mathsf{fail}} \leq 2^{-64}$.

5.2.2 Plaintext-Ciphertext Operation Benchmarks

Next, we consider benchmarks for the plaintext-ciphertext integer operations. In particular, we present the latencies for a variety of algorithms presented in Section 3 for precisions $\log_2(\Omega) \in \{4, 8, 16, 32, 64, 128, 256\}$. Each benchmark is linked to an individual algorithm (or section) in the text, so that the reader knows which benchmarks correspond to which algorithms. Moreover, the names of the benchmarks match the name used in the TFHE-rs library directly, so that a user can re-run each benchmark on their own machine (or different hardware, for comparison) if desired.

Operation	Reference	$\log_2(\Omega)$ 4	8	16	32	64	128	256
`unsigned_overflowing_scalar_add_parallelized`	Section 3.14	38.2 ms	53.2 ms	56.3 ms	57.0 ms	78.6 ms	105 ms	177 ms
`unsigned_overflowing_scalar_sub_parallelized`	Section 3.14	37.9 ms	53.0 ms	56.9 ms	57.9 ms	79.9 ms	106 ms	178 ms
`unsigned_scalar_add_parallelized`	Section 3.6.2	35.9 ms	51.7 ms	56.2 ms	57.0 ms	78.6 ms	104 ms	172 ms
`unsigned_scalar_bit{and,or,xor}_parallelized`	Algorithm 26	18.1 ms	19.0 ms	19.3 ms	19.7 ms	21.3 ms	23.1 ms	25.0 ms
`unsigned_scalar_div_parallelized`	Section 3.17.2	78.8 ms	137 ms	189 ms	269 ms	451 ms	833 ms	2.15 s
`unsigned_scalar_{eq,ne}_parallelized`	Section 3.4.2	17.0 ms	34.6 ms	35.4 ms	56.0 ms	54.4 ms	75.8 ms	78.1 ms
`unsigned_scalar_{gt,ge,lt,le}_parallelized`	Algorithm 69	14.3 ms	36.7 ms	35.5 ms	54.7 ms	74.0 ms	94.5 ms	138 ms
`unsigned_scalar_mul_parallelized`	Algorithm 60	37.9 ms	75.6 ms	118 ms	161 ms	222 ms	419 ms	1.04 s
`unsigned_scalar_rem_parallelized`	Section 3.17.2	146 ms	268 ms	344 ms	492 ms	719 ms	1.27 s	2.86 s
`unsigned_scalar_{left,right}_shift_parallelized`	Algorithms 45 and 48	19.8 ms	18.9 ms	19.3 ms	19.9 ms	20.6 ms	22.1 ms	24.8 ms
`unsigned_scalar_rotate_{left,right}_parallelized`	Algorithms 46 and 49	20.3 ms	19.3 ms	19.4 ms	19.9 ms	21.4 ms	22.7 ms	24.9 ms
`unsigned_scalar_sub_parallelized`	Section 3.8.1	35.2 ms	54.3 ms	58.0 ms	58.5 ms	80.2 ms	106 ms	175 ms

Table 8: Benchmarks for integer plaintext-ciphertext operations. The parameter set in each case corresponds to `2_2_64` in Table 13. In particular, this means that the failure probability is $\leq 2^{-64}$.

Operation	Reference	$\log_2(\Omega)$ 4	8	16	32	64	128	256
unsigned_overflowing_scalar_add_parallelized	Section 3.14	38.8 ms	55.7 ms	62.9 ms	64.4 ms	89.2 ms	115 ms	189 ms
unsigned_overflowing_scalar_sub_parallelized	Section 3.14	39.9 ms	56.1 ms	62.1 ms	64.8 ms	89.4 ms	114 ms	187 ms
unsigned_scalar_add_parallelized	Section 3.6.2	36.5 ms	55.5 ms	60.4 ms	62.7 ms	87.5 ms	112 ms	180 ms
unsigned_scalar_bit{and, or, xor}_parallelized	Algorithm 26	16.1 ms	18.3 ms	20.5 ms	21.4 ms	22.9 ms	24.6 ms	27.1 ms
unsigned_scalar_div_parallelized	Section 3.17.2	76.9 ms	145 ms	197 ms	274 ms	459 ms	830 ms	2.36 s
unsigned_scalar_{eq,ne}_parallelized	Section 3.4.2	14.9 ms	34.6 ms	34.7 ms	54.9 ms	57.0 ms	80.1 ms	82.3 ms
unsigned_scalar_{ge,gt,le,lt}_parallelized	Algorithm 69	14.8 ms	39.6 ms	37.3 ms	55.0 ms	78.0 ms	99.4 ms	145 ms
unsigned_scalar_mul_parallelized	Algorithm 60	42.0 ms	77.0 ms	127 ms	175 ms	240 ms	457 ms	1.09 s
unsigned_scalar_rem_parallelized	Section 3.17.2	151 ms	277 ms	370 ms	527 ms	770 ms	1.44 s	3.17 s
unsigned_scalar_{left,right}_shift_parallelized	Algorithms 45 and 48	21.1 ms	20.4 ms	20.4 ms	21.3 ms	23.1 ms	24.8 ms	26.8 ms
unsigned_scalar_rotate_{left,right}_parallelized	Algorithms 46 and 49	21.2 ms	19.2 ms	20.6 ms	21.3 ms	22.8 ms	24.6 ms	26.4 ms
unsigned_scalar_sub_parallelized	Section 3.8.1	35.0 ms	56.1 ms	62.2 ms	63.9 ms	88.4 ms	114 ms	185 ms

Table 9: Benchmarks for integer plaintext-ciphertext operations. The parameter set in each case corresponds to `2_2_128` in Table 13. In particular, this means that the failure probability is $\leq 2^{-128}$.

5.2.3 Ciphertext-Ciphertext Operation Benchmark

Next, we consider benchmarks for the ciphertext-ciphertext integer operations. In particular, we present the latencies for a variety of algorithms presented in Section 3 for precisions $\log_2(\Omega) \in \{4, 8, 16, 32, 64, 128, 256\}$. Each benchmark is linked to an individual algorithm (or section) in the text, so that the reader knows which benchmarks correspond to which algorithms. Moreover, the names of the benchmarks match the name used in the TFHE-rs library directly, so that a user can re-run each benchmark on their own machine (or different hardware, for comparison) if desired.

		$\log_2(\Omega)$						
Operation	Reference	4	8	16	32	64	128	256
unsigned_add_parallelized	Algorithm 38	32.1 ms	52.5 ms	58.1 ms	79.9 ms	101 ms	161 ms	173 ms
unsigned_bitnot	Algorithm 25	3.19 μs	7.26 μs	14.2 μs	26.7 μs	52.3 μs	100 μs	204 μs
unsigned_bit{and,or,xor}	Algorithm 24	18.9 ms	18.9 ms	19.7 ms	20.3 ms	21.8 ms	22.9 ms	25.0 ms
unsigned_div_rem_parallelized	Algorithm 71	292 ms	667 ms	1.49 s	3.39 s	7.87 s	19.6 s	52.1 s
unsigned_if_then_else_parallelized	Algorithm 63	28.5 ms	28.9 ms	29.4 ms	31.0 ms	31.9 ms	34.3 ms	49.2 ms
unsigned_{left, right}_shift_parallelized	Algorithm 30	19.7 ms	59.9 ms	79.2 ms	100 ms	128 ms	169 ms	235 ms
unsigned_{lt, le, gt, ge}_parallelized	Algorithm 69	36.6 ms	36.1 ms	55.6 ms	74.5 ms	94.8 ms	136 ms	166 ms
unsigned_mul_parallelized	Algorithm 58	38.5 ms	94.4 ms	136 ms	210 ms	381 ms	1.1 s	3.65 s
unsigned_{eq,ne}_parallelized	Algorithm 27	36.9 ms	36.3 ms	55.6 ms	55.0 ms	76.3 ms	77.6 ms	99.9 ms
unsigned_neg_parallelized	Algorithm 39	32.1 ms	48.7 ms	57.0 ms	78.0 ms	103 ms	162 ms	183 ms
unsigned_overflowing_add_parallelized	Section 3.14	34.8 ms	56.5 ms	58.7 ms	79.2 ms	101 ms	162 ms	173 ms
unsigned_overflowing_mul_parallelized	Section 3.14	108 ms	151 ms	181 ms	268 ms	509 ms	1.45 s	4.94 s
unsigned_overflowing_sub_parallelized	Section 3.14	38.9 ms	56.6 ms	55.3 ms	79.2 ms	100 ms	166 ms	178 ms
unsigned_rotate_{left, right}_parallelized	Section 3.9	19.2 ms	57.5 ms	77.1 ms	98.9 ms	132 ms	179 ms	245 ms
unsigned_sub_parallelized	Algorithm 40	36.0 ms	56.4 ms	58.8 ms	80.3 ms	102 ms	162 ms	174 ms
unsigned_sum_ciphertexts_parallelized_5_ctxts	Algorithm 57	37.7 ms	77.2 ms	76.9 ms	97.8 ms	121 ms	188 ms	223 ms
unsigned_sum_ciphertexts_parallelized_10_ctxts	Algorithm 57	58.4 ms	95.5 ms	99.8 ms	123 ms	149 ms	237 ms	297 ms
unsigned_sum_ciphertexts_parallelized_20_ctxts	Algorithm 57	76.4 ms	117 ms	119 ms	146 ms	193 ms	288 ms	419 ms

Table 10: Benchmarks for integer ciphertext-ciphertext operations. The parameter set in each case corresponds to `2_2_64` in Table 13. In particular, this means that the failure probability is $\leq 2^{-64}$.

		$\log_2(\Omega)$						
Operation	Algorithm / Section	4	8	16	32	64	128	256
unsigned_add_parallelized	Algorithm 38	42.6 ms	62.8 ms	59.2 ms	85.5 ms	105 ms	174 ms	190 ms
unsigned_bitnot	Algorithm 25	3.37 μs	5.35 μs	10.3 μs	19.8 μs	39.1 μs	83.9 μs	184 μs
unsigned_bit{and,or,xor}_parallelized	Algorithm 24	22.8 ms	20.9 ms	20.2 ms	21.3 ms	22.7 ms	24.8 ms	27.3 ms
unsigned_div_rem_parallelized	Algorithm 71	293 ms	684 ms	1.52 s	3.51 s	8.42 s	21.0 s	54.6 s
unsigned_if_then_else_parallelized	Algorithm 63	27.9 ms	30.6 ms	31.4 ms	33.7 ms	34.5 ms	38.0 ms	49.6 ms
unsigned_{left, right}_shift_parallelized	Algorithm 30	20.4 ms	59.5 ms	83 ms	109 ms	141 ms	186 ms	255 ms
unsigned_{lt, le, gt, ge}_parallelized	Algorithm 69	39.1 ms	36.9 ms	56.3 ms	77.8 ms	101 ms	147 ms	177 ms
unsigned_mul_parallelized	Algorithm 58	41.5 ms	101 ms	146 ms	219 ms	400 ms	1.13 s	3.77 s
unsigned_{eq, ne}_parallelized	Algorithm 27	35.7 ms	35.9 ms	54.9 ms	57.4 ms	79.1 ms	81.5 ms	105 ms
unsigned_neg_parallelized	Algorithm 39	33.4 ms	48.8 ms	62.6 ms	84.6 ms	106 ms	171 ms	189 ms
unsigned_overflowing_add_parallelized	Section 3.14	40.5 ms	59.2 ms	63.6 ms	86.1 ms	107 ms	175 ms	188 ms
unsigned_overflowing_mul_parallelized	Section 3.14	111 ms	153 ms	181 ms	274 ms	526 ms	1.51 s	5.13 s
unsigned_overflowing_sub_parallelized	Section 3.14	39.1 ms	58.3 ms	61.7 ms	84.7 ms	109 ms	172 ms	192 ms
unsigned_rotate_{left, right}_parallelized	Section 3.9	20.4 ms	61.1 ms	83.0 ms	110 ms	142 ms	191 ms	258 ms
unsigned_sub_parallelized	Algorithm 40	39.9 ms	59.5 ms	60.1 ms	83.3 ms	110 ms	176 ms	191 ms
unsigned_sum_ciphertexts_parallelized_5_ctxts	Algorithm 57	39.0 ms	73.6 ms	82.2 ms	106 ms	134 ms	199 ms	237 ms
unsigned_sum_ciphertexts_parallelized_10_ctxts	Algorithm 57	57.8 ms	98.4 ms	108 ms	130 ms	161 ms	250 ms	319 ms
unsigned_sum_ciphertexts_parallelized_20_ctxts	Algorithm 57	83.5 ms	124 ms	130 ms	156 ms	206 ms	314 ms	432 ms

Table 11: Benchmarks for integer ciphertext-ciphertext operations. The parameter set in each case corresponds to `2_2_128` in Table 13. In particular, this means that the failure probability is $\leq 2^{-128}$.

5.3 Compression and Decompression Benchmarks

In this section, we outline the benchmarks for the compression (Algorithm 73) and decompression (Algorithm 74) algorithms.

			$\log_2(\Omega)$						
Operation	Algorithm	p_{fail}	8	16	32	64	128	256	512
Compress	73	2^{-64}	1.74 ms	1.99 ms	2.42 ms	3.33 ms	4.78 ms	5.89 ms	9.57 ms
Decompress	74	2^{-64}	16.9 ms	17.1 ms	17.5 ms	17.9 ms	18.6 ms	22.0 ms	39.6 ms
Compress	73	2^{-128}	2.72 ms	2.15 ms	2.59 ms	3.46 ms	4.88 ms	5.89 ms	9.65 ms
Decompress	74	2^{-128}	16.7 ms	17.8 ms	18.0 ms	18.4 ms	18.9 ms	22.3 ms	41.9 ms

Table 12: Benchmarks of compression and decompression operations applied to homomorphic integers of various sizes, evaluated for two different failure probabilities.

5.4 Parameters

In Tables 13 and 14, we present all parameter sets considered as part of the benchmarks, listed alongside their respective failure probabilities.

As before, the naming convention of the parameter sets informs the reader about the carry modulus, message modulus and the failure probability. In particular, a classical parameter set `X_X_Y` represents homomorphic integers with a carry modulus of $\mathsf{carry} = 2^{\mathsf{X}}$ and a message modulus of $\mathsf{msg} = 2^{\mathsf{X}}$, with a failure probability of $2^{-\mathsf{Y}}$. For the Multibit parameter sets `X_X_Y_MultBitW`, `W` corresponds to the grouping factor.

Parameter Set	LWE Dimension (n)	LWE Noise (σ_n)	GLWE Dimension (k)	Polynomial Size (N)	GLWE Noise (σ_N)	PBS Base Log (β_{PBS})	PBS Level (ℓ_{PBS})	KS Base Log ($\mathfrak{B}_{\mathsf{KS}}$)	KS Level (β_{KS})	Total Encryptions of Zero (Z)	Avg Number of Zeros (avg)	Probability of failure (p_{fail})
`1_1_40`	750	1.514×10^{-5}	3	512	1.952×10^{-11}	17	1	4	3	-	-	$2^{-40.004}$
`2_2_40`	796	6.846×10^{-6}	1	2048	2.845×10^{-15}	23	1	3	5	-	-	$2^{-40.489}$
`3_3_40`	925	7.393×10^{-7}	1	8192	2.168×10^{-19}	15	2	3	6	-	-	$2^{-40.298}$
`4_4_40`	1096	3.868×10^{-8}	1	32768	2.168×10^{-19}	15	2	4	5	-	-	$2^{-40.107}$
`1_1_64`	781	8.868×10^{-6}	4	512	2.845×10^{-15}	23	1	4	3	-	-	$2^{-64.01}$
`2_2_64`	833	3.616×10^{-6}	1	2048	2.845×10^{-15}	23	1	3	5	-	-	$2^{-64.014}$
`3_3_64`	977	3.014×10^{-7}	1	8192	2.168×10^{-19}	15	2	3	6	-	-	$2^{-64.177}$
`4_4_64`	1110	3.038×10^{-8}	1	32768	2.168×10^{-19}	11	3	2	11	-	-	$2^{-64.014}$
`1_1_128`	838	3.317×10^{-6}	4	512	2.845×10^{-15}	23	1	5	3	1444	17	$2^{-128.979}$
`2_2_128`	866	2.046×10^{-6}	1	2048	2.845×10^{-15}	23	1	3	5	1446	17	$2^{-128.839}$
`3_3_128`	1007	1.796×10^{-7}	1	8192	2.168×10^{-19}	15	2	3	7	1455	17	$2^{-128.857}$
`4_4_128`	1098	3.737×10^{-8}	1	65536	2.168×10^{-19}	11	3	3	7	2961	34	$2^{-128.676}$

Table 13: Typical (Gaussian) computation parameter sets used inside TFHE-rs.

Parameter Set	Grouping factor (gf)	LWE Dimension (n)	LWE Noise (σ_n)	GLWE Dimension (k)	Polynomial Size (N)	GLWE Noise (σ_N)	PBS Base Log (β_{PBS})	PBS Level (ℓ_{PBS})	KS Base Log ($\mathfrak{B}_{\mathsf{KS}}$)	KS Level (β_{KS})	Probability of failure (p_{fail})
1_1_64_MultBit2	2	748	1.567×10^{-5}	2	1024	2.845×10^{-15}	22	1	4	3	$2^{-66.162}$
1_1_64_MultBit3	3	747	1.594×10^{-5}	2	1024	2.845×10^{-15}	22	1	4	3	$2^{-64.618}$
1_1_64_MultBit4	4	712	2.916×10^{-5}	1	2048	2.845×10^{-15}	22	1	3	4	$2^{-65.871}$
2_2_64_MultBit2	2	872	1.845×10^{-6}	1	2048	2.845×10^{-15}	22	1	4	4	$2^{-64.148}$
2_2_64_MultBit3	3	912	9.252×10^{-7}	1	2048	2.845×10^{-15}	22	1	5	3	$2^{-64.095}$
2_2_64_MultBit4	4	872	1.845×10^{-6}	1	2048	2.845×10^{-15}	22	1	4	4	$2^{-64.122}$
3_3_64_MultBit2	2	978	2.963×10^{-7}	1	8192	2.168×10^{-19}	14	2	3	6	$2^{-64.212}$
3_3_64_MultBit3	3	978	2.963×10^{-7}	1	8192	2.168×10^{-19}	14	2	3	6	$2^{-64.232}$
3_3_64_MultBit4	4	980	2.862×10^{-7}	1	8192	2.168×10^{-19}	14	2	3	6	$2^{-64.633}$
4_4_64_MultBit2	2	1122	2.470×10^{-8}	1	32768	2.168×10^{-19}	8	4	2	11	$2^{-64.021}$
4_4_64_MultBit3	3	1119	2.601×10^{-8}	1	32768	2.168×10^{-19}	8	4	2	11	$2^{-64.022}$
4_4_64_MultBit4	4	1124	2.386×10^{-8}	1	32768	2.168×10^{-19}	8	4	2	11	$2^{-64.007}$

Table 14: Typical (Gaussian, Multi-bit) computation parameter sets used inside TFHE-rs.

In Table 15 we present the parameters used for the compression and decompression algorithm benchmarks outlined in Table 12. `COMPRESS_X` corresponds to the parameters used for compression with a failure probability of $2^{-\mathsf{X}}$

Parameter Set	BR Base Log ($\mathfrak{B}_{\mathsf{br}}$)	BR Level (ℓ_{br})	PKS Base Log ($\mathfrak{B}_{\mathsf{pks}}$)	PKS level (ℓ_{pks})	Polynomial Size (N)	GLWE Noise (σ_N) $\lceil (B) \rfloor$	GLWE Dimension (k)	Number of LWEs (α)	Storage Modulus (q_{stg})	Probability of failure (p_{fail})
`COMPRESS_64`	1	23	2	6	256	1.340×10^{-15}	4	256	12	$\leq 2^{-64}$
`COMPRESS_128`	1	23	2	6	256	1.340×10^{-15}	4	256	12	$\leq 2^{-128}$

Table 15: Typical compression parameters used inside TFHE-rs.

For completeness, in Table 16, we present typical parameters used in TFHE-rs when using a TUniform noise distribution.

Parameter Set	LWE Dimension (n)	LWE Noise (B_n)	GLWE Dimension (k)	Polynomial Size (N)	GLWE Noise (B_N)	PBS Base Log (β_{PBS})	PBS Level (ℓ_{PBS})	KS Base Log ($\mathfrak{B}_{\mathsf{KS}}$)	KS Level (β_{KS})	Total Encryptions of Zero (Z)	Avg Number of Zeros (avg)	Probability of failure (p_{fail})
`1_1_40`$_\mathsf{T}$	799	48	3	512	30	17	1	3	4	-	-	$2^{-40.525}$
`2_2_40`$_\mathsf{T}$	839	47	1	2048	17	23	1	3	5	-	-	$2^{-57.015}$
`3_3_40`$_\mathsf{T}$	958	44	1	8192	3	15	2	3	6	-	-	$2^{-50.002}$
`4_4_40`$_\mathsf{T}$	1077	41	1	32768	3	15	2	3	7	-	-	$2^{-41.009}$
`1_1_64`$_\mathsf{T}$	839	47	4	512	17	23	1	4	3	-	-	$2^{-72.226}$
`2_2_64`$_\mathsf{T}$	879	46	1	2048	17	23	1	3	5	-	-	$2^{-72.178}$
`3_3_64`$_\mathsf{T}$	998	43	1	8192	3	15	2	3	6	-	-	$2^{-64.454}$
`4_4_64`$_\mathsf{T}$	1117	40	1	32768	3	11	3	1	22	-	-	$2^{-64.037}$
`1_1_128`$_\mathsf{T}$	879	46	4	512	17	23	1	5	3	1437	15	$2^{-144.044}$
`2_2_128`$_\mathsf{T}$	918	45	1	2048	17	23	1	4	4	1449	17	$2^{-129.153}$
`3_3_128`$_\mathsf{T}$	1077	41	1	8192	3	15	2	4	5	1459	17	$2^{-128.771}$
`4_4_128`$_\mathsf{T}$	1117	40	1	65536	3	11	3	3	7	2948	31	$2^{-141.493}$

Table 16: Typical (TUniform) computation parameter sets used inside TFHE-rs.

6 Find Out More

By explaining the algorithms and integer arithmetic of TFHE-rs, this document aims to support developers building backends for homomorphic evaluations based on the TFHE scheme in general, and more specifically, the variant implemented in TFHE-rs. However, as advancements in the FHE space might raise additional questions, more supporting resources can be found through the following links:

- additional information on the TFHE-rs library: https://docs.zama.ai/tfhe-rs
- codebase of the TFHE-rs library: https://github.com/zama-ai/tfhe-rs
- general information on the TFHE scheme: https://www.tfhe.com/about
- support channels for all technical questions: https://community.zama.ai/ and https://discord.com/invite/zama
- contact point for collaborations or any other inquiries: hello@zama.ai

References

[ACPS09] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In Shai Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 595–618, Santa Barbara, CA, USA, August 16–20, 2009. Springer Berlin Heidelberg, Germany.

[AP14] Jacob Alperin-Sheriff and Chris Peikert. Faster bootstrapping with polynomial error. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 297–314, Santa Barbara, CA, USA, August 17–21, 2014. Springer Berlin Heidelberg, Germany.

[BBB+23] Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Parameter optimization and larger precision for (T)FHE. *Journal of Cryptology*, 36(3):28, July 2023.

[BCK+23] Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, Jai Hyun Park, and Damien Stehlé. HERMES: Efficient ring packing using MLWE ciphertexts and application to transciphering. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023, Part IV*, volume 14084 of *Lecture Notes in Computer Science*, pages 37–69, Santa Barbara, CA, USA, August 20–24, 2023. Springer, Cham, Switzerland.

[BGGJ20] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. CHIMERA: combining ring-lwe-based fully homomorphic encryption schemes. *J. Math. Cryptol.*, 14(1), 2020.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, 2012.

[BIP+22a] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. Final: Faster fhe instantiated with ntru and lwe. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022*, pages 188–215, Cham, 2022. Springer Nature Switzerland.

[BIP+22b] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. FINAL: Faster FHE instantiated with NTRU and LWE. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part II*, volume 13792 of *Lecture Notes in Computer Science*, pages 188–215, Taipei, Taiwan, December 5–9, 2022. Springer, Cham, Switzerland.

[BJRW23] Katharina Boudgoust, Corentin Jeudy, Adeline Roux-Langlois, and Weiqiang Wen. On the hardness of module learning with errors with short distributions. *Journal of Cryptology*, 36(1):1, January 2023.

[BJSW24] Olivier Bernard, Marc Joye, Nigel P. Smart, and Michael Walter. Drifting towards better error probabilities in fully homomorphic encryption schemes. Cryptology ePrint Archive, Paper 2024/1718, 2024.

[BMMP18] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of

Lecture Notes in Computer Science, pages 483–512, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Cham, Switzerland.

[BST20] Florian Bourse, Olivier Sanders, and Jacques Traoré. Improved secure integer comparison via homomorphic encryption. In *CT-RSA*. Springer, 2020.

[CDKS21] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient homomorphic conversion between (ring) LWE ciphertexts. In Kazue Sako and Nils Ole Tippenhauer, editors, *ACNS 21: 19th International Conference on Applied Cryptography and Network Security, Part I*, volume 12726 of *Lecture Notes in Computer Science*, pages 460–479, Kamakura, Japan, June 21–24, 2021. Springer, Cham, Switzerland.

[CGGI16a] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33, Hanoi, Vietnam, December 4–8, 2016. Springer Berlin Heidelberg, Germany.

[CGGI16b] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology - ASIACRYPT 2016*, 2016.

[CGGI16c] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption library. https://tfhe.github.io/tfhe/, August 2016.

[CGGI17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In *Advances in Cryptology - ASIACRYPT 2017*, 2017.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020. Earlier versions in ASIACRYPT 2016 and 2017.

[CJL+20] Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Concrete: Concrete operates on ciphertexts rapidly by extending TfhE. In *WAHC 2020*, 2020.

[CJP21] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *Cyber Security Cryptography and Machine Learning - 5th International Symposium, CSCML 2021*, volume 12716 of *Lecture Notes in Computer Science*. Springer, 2021.

[CLOT21] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe. In *ASIACRYPT 2021*. Springer, 2021.

[CZB+22] Pierre-Emmanuel Clet, Martin Zuber, Aymen Boudguiga, Renaud Sirdey, and Cédric Gouy-Pailler. Putting up the swiss army knife of homomorphic calculations by means of tfhe functional bootstrapping. Cryptology ePrint Archive, Report 2022/149, 2022. https://ia.cr/2022/149.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 617–640. Springer, 2015.

[GBA21] Antonio Guimarães, Edson Borin, and Diego F. Aranha. Revisiting the functional bootstrap in TFHE. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2), 2021.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 169–178, Bethesda, MD, USA, May 31 – June 2, 2009. ACM Press.

[GINX16] Nicolas Gama, Malika Izabachène, Phong Q. Nguyen, and Xiang Xie. Structural lattice reduction: Generalized worst-case to average-case reductions and homomorphic cryptosystems. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 528–558, Vienna, Austria, May 8–12, 2016. Springer Berlin Heidelberg, Germany.

[GM94] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 6172, New York, NY, USA, 1994. Association for Computing Machinery.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2013.

[HSJ86] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[Joy21] Marc Joye. Balanced non-adjacent forms. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part III*, volume 13092 of *Lecture Notes in Computer Science*, pages 553–576. Springer, 2021.

[Joy24] Marc Joye. TFHE public-key encryption revisited. In Elisabeth Oswald, editor, *Topics in Cryptology – CT-RSA 2024*, volume 14643 of *Lecture Notes in Computer Science*, pages 277–291, San Francisco, CA, USA, May 6–9, 2024. Springer, Cham, Switzerland.

[JP22] Marc Joye and Pascal Paillier. Blind rotation in̆fully homomorphic encryption with̆extended keys. In Shlomi Dolev, Jonathan Katz, and Amnon Meisels, editors, *Cyber Security, Cryptology, and Machine Learning*, pages 1–18, Cham, 2022. Springer International Publishing.

[KÖ22] Jakub Klemsa and Melek Önen. Parallel operations over TFHE-encrypted multi-digit integers. In *Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy*, pages 288–299, 2022.

[LATV12] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. *Proceedings of the Annual ACM Symposium on Theory of Computing*, 05 2012.

[Lee24] Yongwoo Lee. Lmkcdey revisited: Speeding up blind rotation with signed evaluation keys. *Mathematics*, 12(18), 2024.

[Lib24] Benoît Libert. Vector commitments with̆proofs of̆smallness: Short range proofs and̆more. In Qiang Tang and Vanessa Teague, editors, *Public-Key Cryptography – PKC 2024*, pages 36–67, Cham, 2024. Springer Nature Switzerland.

[LMK+23] Yongwoo Lee, Daniele Micciancio, Andrey Kim, Rakyong Choi, Maxim Deryabin, Jieun Eom, and Donghoon Yoo. Efficient FHEW bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part III*, volume 14006 of *Lecture Notes in Computer Science*, pages 227–256, Lyon, France, April 23–27, 2023. Springer, Cham, Switzerland.

[LMP21] Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. Large-precision homomorphic sign evaluation using ffhew/tfhe bootstrapping. Cryptology ePrint Archive, Report 2021/1337, 2021. https://ia.cr/2021/1337.

[LMSS23] Changmin Lee, Seonhong Min, Jinyeong Seo, and Yongsoo Song. Faster TFHE bootstrapping with block binary keys. In Joseph K. Liu, Yang Xiang, Surya Nepal, and Gene Tsudik, editors, *ASIACCS 23: 18th ACM Symposium on Information, Computer and Communications Security*, pages 2–13, Melbourne, VIC, Australia, July 10–14, 2023. ACM Press.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*. Springer, 2010.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

[MM11] Daniele Micciancio and Petros Mol. Pseudorandom knapsacks and the sample complexity of LWE search-to-decision reductions. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 465–484, Santa Barbara, CA, USA, August 14–18, 2011. Springer Berlin Heidelberg, Germany.

[MP12] Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 700–718, Cambridge, UK, April 15–19, 2012. Springer Berlin Heidelberg, Germany.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, 2005*. ACM, 2005.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):34:1–34:40, 2009. Earlier version in STOC 2005.

[SSTX09] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 617–635. Springer, 2009.

[vDGHV10] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 24–43, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[WWL+24] Ruida Wang, Yundi Wen, Zhihao Li, Xianhui Lu, Benqiang Wei, Kun Liu, and Kunpeng Wang. Circuit bootstrapping: Faster and smaller. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024, Part II*, volume 14652 of *Lecture Notes in Computer Science*, pages 342–372, Zurich, Switzerland, May 26–30, 2024. Springer, Cham, Switzerland.

[XZD+23] Binwu Xiang, Jiang Zhang, Yi Deng, Yiran Dai, and Dengguo Feng. Fast blind rotation for bootstrapping FHEs. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023, Part IV*, volume 14084 of *Lecture Notes in Computer Science*, pages 3–36, Santa Barbara, CA, USA, August 20–24, 2023. Springer, Cham, Switzerland.

[ZYL+18] Tanping Zhou, Xiaoyuan Yang, Longfei Liu, Wei Zhang, and Ningbo Li. Faster bootstrapping with multiple addends. *IEEE Access*, 6:49868–49876, 2018.