Parameter Optimization and Larger Precision for (T)FHE

Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, Samuel Tap

Zama, Paris, France - https://zama.ai/
{loris.bergerat, anas.boudi, quentin.bourgerie, ilaria.chillotti, damien.ligier, jb.orfila, samuel.tap}@zama.ai

June 15, 2022

Abstract

In theory, Fully Homomorphic Encryption schemes allow to compute any operation over encrypted data. However in practice, one of the major difficulties lies into determining secure cryptographic parameters that reduce the computational cost of evaluating a circuit. In this paper, we propose a framework of optimization to solve this open problem. Even though it mainly focuses on TFHE, the method is generic enough to be adapted to any FHE scheme. As an application, this framework allows us to design solutions to efficiently increase the precision initially supported by the TFHE scheme to large integers. Beyond the classical radix encoding of plaintexts, we propose an alternative representation making use of the Chinese Remainder Theorem, which is particularly suited for parallel computation. We show how to evaluate operations on these new ciphertext types, from basic arithmetic operations, to more complex ones, such as the evaluation of a generic look-up table. The latter relies on a new efficient way to evaluate a programmable bootstrapping. Finally, we propose a plethora of applications of the optimization framework, such as true comparisons between bootstrapping operators, i.e. not only on the computation time but also on the amount of output error and more importantly the probability of failure all at once.

Contents

1 Introduction

Fully Homomorphic Encryption (FHE) refers to an encryption scheme that allows to perform a potentially unlimited amount of computations over encrypted data. FHE schemes have attracted a lot of attention in the last decade. Indeed, they could solve many real world applications on which the privacy of the data manipulated has to be preserved. The ones that are mainly studied nowadays are based on hard problems over lattices: LWE [Reg05], and its variant RLWE [LPR10, SSTX09]. We can mention the following schemes: BGV [BGV12], B/FV [Bra12, FV12], HEAAN [CKKS17], GSW [GSW13], FHEW [DM15] and TFHE [CGGI20].

One of the main problems of FHE schemes is to find parameters that are both secure and that make the operations as efficient as possible, in terms of both computational cost and memory. Solving this problem is fundamental if we plan for a large scale adoption of FHE schemes. Regarding the security constraints, the LWE/Lattice-estimator [APS15] is the main tool to evaluate the security of the parameters for an LWE-based cryptosystem. However, it does not help finding efficient parameters for a given context. Finding the optimal parameter set is even harder and no solution has been proposed yet.

In this paper, we mainly focus on the TFHE scheme [CGGI20]. The latter is particularly interesting because it offers a bootstrapping technique that is able to reduce the noise of ciphertexts and, at the same time, to evaluate a function on the input, expressed as a Look-Up Table (LUT). This is often called programmable bootstrapping (PBS). Unfortunately, in practice, the bootstrapping takes as input a single LWE ciphertext encrypting a small message (say at most 8 bits). To evaluate operations on large precision messages, the only way is to split the message in many ciphertexts and then build the circuit to evaluate as a combination of linear operations and ciphertexts. The solutions presented until now, however, are not very efficient.

Our contributions We introduce the **first optimization framework** for FHE computations in Section 3. Roughly speaking, it is a generic approach taking as input a graph of mathematical operators, such as additions, multiplications or LUT evaluations. Integer values across this graph have some metadata regarding their precision and attributes distinguishing data that should be encrypted. The output is an optimal graph of FHE operators along with the optimal parameter set for this graph.

We started to instantiate our framework for many applications that are also described in this paper. Indeed, to the best of our knowledge, we propose the first actual solution to optimize TFHE-like scheme parameters, automatizing a crucial process to see FHE adopted massively. It also allows to compare different algorithms executed with their optimal parameters in a fair way, because it takes into account the probability of failure and the output noise.

As one application of the optimizer, we also propose to overcome one of the major limitation of the TFHE scheme: efficiently extending homomorphic computations

over inputs of **precision larger than** 8 **bits** (Section 4). We start by defining a new representation of small precision ciphertexts (smaller than 8 bits) so that it includes a carry buffer. This allows to efficiently compute leveled operation without having to compute bootstraps after each one of them. Then, we build new extended ciphertext types: one based on a classical radix decomposition, and another one based on the CRT decomposition of plaintexts. As far as we know, the latter strategy has never been described before, particularly for the TFHE scheme. We additionally propose a new WoP-PBS technique (PBS without bit of padding) allowing to evaluate LUTs on these new type of ciphertexts. We show that this approach is better than the already existing ones when the precision becomes larger, such as the one in [LMP21].

Finally, we present a number of **practical applications** that benefit from our new optimization framework, including experiments on our new ciphertext types, or the comparison between homomorphic operators (Section 5).

Related works on optimization In the literature, a few compilers for FHE schemes have been proposed: their major optimization goal is in terms of circuit shape, i.e. they try to change the order and type of operations to make execution more efficient. When it comes to finding parameters for TFHE, existing compilers [CMG+18, CDS15] do not look for the best parameter set, they always use the same ones and focus on optimizing the Boolean circuit instead.

For schemes such as B/FV or CKKS, that have the tendency to avoid bootstrapping and favor leveled operations, the existing compilers do this parameter selection. For TFHE-like schemes, the circuits evaluated by these compilers are binary circuits, using gate bootstrapping, with hard-coded parameters. We suggest this paper [VJH21] for more information and for comparisons on all existing FHE compilers. To the best of our knowledge, no one has ever presented a result on optimization of parameters for an FHE scheme (including bootstrapping) with a flexibility for multi-precision plaintexts.

Related works on homomorphic integers Apart from the binary approach proposed in FHEW [DM15] and TFHE [CGGI20], encrypting the binary representation of the input one bit per ciphertext, the idea of splitting a message into multiple ciphertexts has already been proposed in [BST20], [GBA21], [KO22], [CZB+22], [LMP21] and [CLOT21]. However, none of them takes advantage of carry buffers to make the computations more efficient between multi-ciphertext encrypted integers and to avoid bootstrappings. In [GBA21] they propose two approaches to evaluate the PBS over these multi-ciphertexts inputs, called *tree-based* and *chained-based* approaches (that we shorten by TreePBS and ChainedPBS). The ChainedPBS method is generalized in [CZB+22] to any function in exchange of a larger plaintext space. In [CLOT21], the authors propose for the first time a WoP-PBS technique, i.e., a PBS that does not require a bit of padding. After them, two different WoP-PBS were proposed in [LMP21] and [KS21].

Even if in the literature there exist many solutions that use a CRT approach on ciphertexts to improve computations for BGV, B/FV and HEAAN, to the best of

our knowledge no work in the state of the art has ever studied a CRT based solution for plaintexts.

2 Notations & Background

In this paper we use the following acronyms: AP refers to atomic pattern, BR to blind rotation, BSK to bootstrapping key, CDF to cumulative distribution function, CRT to Chinese remainder theorem, DAG to directed acyclic graph, DP to dot product, FHE to fully homomorphic encryption, KS to key switch, KSK to key switching key, MS to modulus switching, MSB to most significant bit(s), LSB to least significant bit(s), LUT to lookup table, PBS to programmable bootstrapping, SE to sample extraction, and WoP-PBS to without padding programmable bootstrapping.

We start with background about probabilities and homomorphic encryption.

Definition 1 (Standard score) *Let $A \hookleftarrow \mathcal{N}(\mu, \sigma^2)$ (normal distribution), let p_{err} be an error probability and let Φ be the CDF of A. We define the* standard score z^* *for p_{err} as $z^*(p_{err}) = \Phi^{-1}(1 - \frac{p_{err}}{2}) = -\Phi^{-1}(\frac{p_{err}}{2})$ and we have: $\mathbb{P}(A \notin [\mu - z^*\sigma, \mu + z^*\sigma]) < p_{err}$.*

Theorem 1 (Confidence Interval of a Centered Normal Distribution) *Let $A \hookleftarrow \mathcal{N}(0, \sigma^2)$, $t \in \mathbb{R}$ and $p_{err} \in [0, 1]$. Let $z^*(p_{err})$, the standard score for p_{err}, we have: $z^*(p_{err})\sigma < t \Rightarrow \mathbb{P}(A \notin [-t, t] < p_{err})$.*

The security of TFHE-like schemes is based on the hardness of the LWE problem and its variants. There are several types of ciphertexts used in TFHE. We start by defining the encoding function that we use in this entire paper for LWE ciphertexts.

Definition 2 (GLWE Encode & Decode) *Let $q \in \mathbb{N}$ be a ciphertext modulus, and let $p \in \mathbb{N}$ an integer for the message modulus, and $\pi \in \mathbb{N}$ the number of bit of padding. We have $2^\pi \cdot p \leq q$ and $2^\pi \cdot p$ is the plaintext modulus. Let $m \in \mathbb{Z}_p$ be a message and let $\Delta = \frac{q}{2^\pi \cdot p} \in \mathbb{Q}$ be the scaling factor. We define the encoding of m as: $\tilde{m} = \mathsf{Encode}(m, 2^\pi \cdot p, q) = \lfloor \Delta \cdot m \rceil \in \mathbb{Z}_q$. To decode, we compute the following function: $m = \mathsf{Decode}(\tilde{m}, 2^\pi \cdot p, q) = \lfloor \frac{\tilde{m}}{\Delta} \rceil \in \mathbb{Z}_{2^\pi \cdot p}$.*

Note that p and q do not have to be powers of two and that we overload these functions for polynomials: $\tilde{R} = \mathsf{Encode}(R, 2^\pi \cdot p, q) = \sum_{i=0}^{N-1} \mathsf{Encode}(r_i, 2^\pi \cdot p, q) \cdot X^i$ where $R = \sum_{i=0}^{N-1} r_i \cdot X^i$ and it works the same for Decode.

We recall below the definition of GLWE, RLWE and LWE ciphertexts.

Definition 3 (GLWE Ciphertext) *Given a message $M \in \mathfrak{R}_q = \mathbb{Z}_q[X]/(X^N + 1)$ and a secret key $\vec{S} = (S_1, \cdots, S_k) \in \mathfrak{R}_q^k$, with coefficients either sampled from a uniform binary, uniform ternary or Gaussian distribution, a GLWE ciphertext of M under the secret key $\vec{S}$ is defined as the tuple:*

$$\mathsf{CT} = \left(A_1, \cdots, A_k, B = \sum_{i=1}^{k} A_i \cdot S_i + \tilde{M} + E \right) = \mathsf{CT}_{\vec{S}}(\tilde{M}) \in \mathfrak{R}_q^{k+1}$$

such that $\{A_i\}_{i=1}^k$ *are polynomials in* $\mathfrak{R}_q$ *with coefficients sampled from the uniform distribution in* $\mathbb{Z}_q$*,* E *is a noise (error) polynomial in* $\mathfrak{R}_q$*, with coefficients sampled from a Gaussian distributions* χ_σ*, and with* $\tilde{M} = \mathsf{Encode}(M, p, q)$*. The parameter* $k \in \mathbb{Z}_{>0}$ *represents the number of polynomials in the GLWE secret key. To simplify notations, we sometimes define* S_{k+1} *as* -1*.*

A GLWE ciphertext with $N = 1$ is called **LWE ciphertext**: in this case we note the size of the secret key by $n = k$, and we note both the ciphertext and the secret with a lower case, e.g. ct and $\vec{s}$. A GLWE ciphertext with $k = 1$ and $N > 1$ is called **RLWE ciphertext**.

In TFHE-like schemes, another type of ciphertext is used, and it is called GGSW (Generalized GSW [GSW13]). A GGSW ciphertext is composed of $(k+1)\ell$ GLWE ciphertexts, encrypting the same message times elements of the secret key with some redundancy. The redundancy is defined by a decomposition base β and a number of levels ℓ in the decomposition. GGSW ciphertexts are used for bootstrapping keys and in the circuit bootstrapping, later described.

In FHE schemes, the technique used to reduce the noise is called *bootstrapping*. In TFHE-like schemes, bootstrapping is also able to evaluate a LUT at the same time. For this reason it is often called *programmable bootstrapping* [CGGI20, CJL+20, CJP21], or **PBS** in short. The PBS is composed of 3 sequential operators: a modulus switching (MS), a blind rotation (BR) and a sample extraction (SE). A PBS, takes as input a LWE ciphertext $\mathsf{ct_{in}}$ encrypting a message m under a secret uniform binary key $\vec{s} \in \mathbb{Z}^n$, a bootstrapping key BSK encrypting the bits of $\vec{s}$ as GGSW ciphertexts under a secret key $\vec{S'} \in \mathfrak{R}^k$, and a polynomial P_L encoding a r-redundant LUT for $x \mapsto L[x]$. The PBS returns a LWE ciphertext $\mathsf{ct_{out}}$ encrypting $L[m]$ under the secret key $\vec{s'}$, extracted form $\vec{S'}$, with smaller noise (if the parameters are chosen appropriately). There is however a probability of failure where the output is actually $L[m + \epsilon]$ with $\epsilon \neq 0$. The signature of the PBS is: $\mathsf{ct_{out}} \leftarrow \mathsf{PBS}(\mathsf{ct_{in}}, \mathsf{BSK}, P_L)$.

Two additional parameters can be used in the PBS to obtain a generalized PBS as in [CLOT21]: $(\varkappa, \vartheta)$. These two parameters define the exact part of the plaintext that is extracted by modulus switching during the PBS.

An **LWE-to-LWE key switching** is an homomorphic operator allowing to switch the secret key as well as a few parameters, and details can be found in [CGGI20, CLOT21]. It takes as input a LWE ciphertext $\mathsf{ct_{in}}$ encrypting a message m under a secret key $\vec{s'} \in \mathbb{Z}^{n'}$, and a key switching key KSK encrypting $\vec{s'}$ with redundancy under another LWE secret key $\vec{s} \in \mathbb{Z}^n$. It returns a LWE ciphertext $\mathsf{ct_{out}}$ encrypting m under the secret key $\vec{s}$, with a larger noise. Its signature is $\mathsf{ct_{out}} \leftarrow \mathsf{KS}(\mathsf{ct_{in}}, \mathsf{KSK})$. To simplify notation, we note $\mathsf{ct}_{out} \leftarrow \mathsf{KS\text{-}PBS}(\mathsf{ct_{out}}, \mathsf{PUB}, P_f)$ where $\mathsf{PUB} = (\mathsf{BSK}, \mathsf{KSK})$ when a KS is followed by a PBS.

In 2017, Chillotti et al. [CGGI20] proposed an operator called **circuit bootstrapping**, transforming a LWE ciphertext into a GGSW ciphertext. It consists in performing some PBS followed by LWE-to-GLWE KS. The later operator converts an LWE ciphertext encrypting m into a GLWE ciphertext encrypting the constant polynomial m. The authors also proposed two operators to evaluate LUTs in a leveled way, called **horizontal and vertical packing**. They both take as

input a d-bit message msg, encrypted as a list of d GGSW ciphertexts encrypting one of its bits. They also take in input α LUTs $L_0 = [l_{0,0}, \cdots, l_{0,2^d-1}], \dots L_{\alpha-1} = [l_{\alpha-1,0}, \cdots, l_{\alpha-1,2^d-1}]$: the goal is to compute the result of the evaluation of the LUTs on the input message, i.e., return encryptions of $l_{0,\mathsf{msg}}, \dots, l_{\alpha-1,\mathsf{msg}}$. Both operators use CMux gates, either as a tree or in a blind rotation) to compute the LWE result (that can be a GLWE in horizontal packing). Horizontal packing is interesting when many LUTs should be evaluated in parallel, while vertical packing is interesting when a single (large) LUT needs to be evaluated. They are two extremes of a trade-off for the evaluation of homomorphic LUTs: in [CGGI20], a mixed solution has been proposed generalizing them.

In this paper, especially for the optimization part, we will require a few higher level definition. For instance, we formalize what a FHE operator is.

Definition 4 (FHE & Plain Operator) *Any FHE operator $\mathcal{O}$ is an implementation of an FHE algorithm, on a given piece of hardware, taking as input some ciphertexts and/or plaintexts and returning one or more ciphertexts. A plain operator is a function mapping several integers into an output list of integers.*

Definition 5 (Noise & Cost Model) *FHE operators are associated with a noise model, a cost model and an plain operator. A noise model is often a formula used to model the noise evolution across $\mathcal{O}$. The cost model is a surrogate for the metric one wants to minimize, it could be the execution time, the power consumption, or the price. A cost is written $\mathsf{Cost}(\cdot)$.*

Noise formula for a given homomorphic operator takes as input the variance of the input ciphertexts noise, some cryptographic parameters involved in the operator computation, as well as the plaintexts values used in the operator.

In this paper we will always consider the cost model to approximate the running time on a single thread.

The **noise of a freshly encrypted ciphertext** is a random (small) integer drawn from a given distribution $\chi(\sigma)$, where σ^2 is its variance. Variances help us quantifying noise in ciphertext, so whenever it is written that a ciphertext contains more noise than another, we mean that the noise inside the first ciphertext is drawn from a normal distribution with a bigger variance than the second one.

The **security** of a GLWE-based scheme depends on the distribution of the secret key (for example binary, ternary or Gaussian), the product between the GLWE dimension and the polynomial size (i.e. $k \cdot N$), the noise distribution, and the ciphertext modulus (often written q). To estimate the security level offered by some given parameters one can use the LWE/Lattice-estimator. As a general rule of thumb, increasing the product $k \cdot N$ decreases the minimal noise needed inside a ciphertext while keeping the same level of security.

As there is a link between the GLWE dimension, the polynomial size, the variance of the Gaussian noise and the level of security, one of these variables can be computed from the other ones. In the rest of this paper, we assume that, for each possible distributions of the secret key, we have access to an oracle that, given the product

$k \cdot N$ and a level of security λ, outputs the minimal noise variance σ^2_{min} required in a ciphertext to achieve the required level of security.

We know that to decrease the minimal noise, while keeping the same level of security, we need to increase other parameters such as the polynomial size or the GLWE dimension. As almost every FHE operator has a cost that depends on both of them, we have a clear trade-off between noise and cost. We want the noise to be small enough to guarantee the correctness of the computation but at the same time, having a smaller noise than needed, is costly because parameters are bigger.

3 Optimization Framework

We introduce in this section the first optimization framework for homomorphic computations based on TFHE. It allows to solve the problem of finding cryptographic parameters, figuring out an encoding for a given data set and picking the best algorithms for a given use-case, which is known to be extremely hard. We designed a framework modeling this problem, and we started to use it in many different contexts. We first need to define two types of Directed Acyclic Graphs (DAG) that are required in the framework.

Definition 6 (Plain DAG) *Let $\overline{\mathfrak{G}} = \left(\overline{V}, \overline{E}\right)$ be a DAG of plain operators. We define $\overline{V} = \left\{\overline{\mathfrak{O}}_i\right\}_{1 \leq i \leq \alpha}$ as the set of vertices, each of them being a plain operator that can be additions, multiplications, subtractions, LUT evaluation, and many others. We define $\overline{E}$ as the set of edges, each of them associated with the precision p of the message as well as a label which is either "private" or "public". When $\overline{E}$ is not needed, we will simply write $\overline{\mathfrak{G}} = \overline{V}$.*

Definition 7 (FHE DAG) *Let $\mathfrak{G} = (V, E)$ be a DAG of FHE operators. We define $V = \{\mathfrak{O}_i\}_{1 \leq i \leq \alpha}$ as the set of vertices, each of them being an FHE Operator. We define E as the set of edges, each of them associated with the modulus p of the encrypted message. When E is not needed, we will simply write $\mathfrak{G} = V$.*

We note $\mathcal{S}_{\mathsf{FHE}}\left(\overline{\mathfrak{G}}\right)$ the set of all possible FHE graphs computing the same functionality than $\overline{\mathfrak{G}}$, a plain DAG. We note $\mathsf{Cost}\left(\mathfrak{G}, x\right)$ the cost of running the FHE graph $\mathfrak{G}$ with the parameter set x.

The optimization framework takes as input a plain DAG $\overline{\mathfrak{G}}$, a level of security, and a correctness probability. It outputs an FHE DAG $\mathfrak{G}$ as well as a parameter set x for $\mathfrak{G}$. Remember that most of the FHE operators in $\mathfrak{G}$ introduce some cryptographic parameters, for instance a local polynomial size $N \in \mathbb{N}$ or a local base $\beta \in \mathbb{N}$. It implies that the total number of possible parameter sets is exponentially huge and we want x to be the best of them all. Also remember that for a same plain operator, for instance an homomorphic multiplication, there are many possible strategies to translate it into an FHE sub-graph. It contributes to increasing the number of possible different topology of $\mathfrak{G}$, which also expends the space where the best solution lies. Our optimization framework guarantees that using the output parameter set x in the graph $\mathfrak{G}$ one has:

(i) the desired level of security,

(ii) the correctness of the computation, up to the given probability.

(iii) the FHE DAG computes the plain functionality described in the plain DAG.

(iv) the cost of the graph is minimized under some hypothesis.

3.1 Foundations of Our Framework

We start by explaining the core ideas to ensure the three mentioned above guarantees. Then we will define the objects that are manipulated during the optimization process.

Definition 8 (Noise Bound) *We call noise bound a threshold related to the noise inside a given ciphertext. It is an integer value $t_\alpha(\pi, p) \in \mathbb{N}$, that depends on some context parameters: p is the message modulus for encrypted messages, π is the number of bits of padding and $\alpha \in [0, 1] \subset \mathbb{R}$ is the probability of failure. This function is not restricted to the mentioned inputs, for instance it could take the degree of fullness later defined in this paper. Note that sometimes we omit π and α when there is no ambiguity.*

For security reasons, the exact noise value has to be kept secret, however the variance of its distribution is publicly known. To guarantee the correctness of the homomorphic computation, we want $\mathbb{P}\left(|e| > t_\alpha\left(\pi, p\right)\right) \leq \alpha$ for a given probability α.

Guarantee (i) We deal with this guarantee with the *minimal noise oracle* which takes as input the product $k \cdot N$ (i.e. the GLWE dimension times the polynomial size) and the desired level of security λ. It returns the minimal encryption noise required to achieve the desired level of security.

By not considering the encryption noise as a variable of the optimization problem but rather as an output of this oracle, we are sure that the level of security will be at least the requested one.

Guarantee (ii) Regarding the second guarantee about the correctness of the homomorphic computation, one must recall that if the noise grows above the noise bound, the decryption algorithm won't output a correct result with more probability than expected. This threshold can be either given as input of the optimization problem or computed from the ciphertext metadata (such as the number of bits to represent the message) and the probability of success of the decryption we want to achieve as described above. The tightness of the noise model to the real life behavior is crucial to correctly solve the problem.

Guarantee (iii) To guarantee the exact same computation between the two graphs, we have, for each plain operator, a list of FHE sub-graphs computing this very mathematical operation (if correctly parameterized).

Guarantee (iv) This framework ends its optimization process with a solver that finds the optimal solution with respect to the cost model. The more realistic the cost model is, the better the solution will be in practice.

As mentioned before, we deal with an immense search space and a smaller space of parameters that do not cross over the noise bounds which is extremely hard to compute. Let us formalize those sets.

Definition 9 (Search Spaces) *For a given FHE DAG $\mathcal{G}$, all the cryptographic parameters that we need to optimize make the set of possible parameters exponentially*

huge. We call this set the search space $\mathfrak{C}_{\mathfrak{G}}$ *for a given graph* $\mathfrak{G}$ *and it is the Cartesian product of the search spaces of all the operators in the graph. We will omit the* $\mathfrak{G}$ *and simply note it* $\mathfrak{C}$ *if there is no ambiguity on the graph.*

We want to optimize parameters for practical use-cases, so we defined the search space of each operators as the range supported by the FHE library targeted. For example, in our case, we restrict polynomial sizes up to $\leq 2^{14}$.

Definition 10 (Noise Feasible Set) *The* noise feasible set *is the set verifying the noise constraints:* $\mathcal{S}_{\mathbb{N}}(\mathfrak{G}) = \{x \in \mathfrak{C} | \forall i, \mathbb{N}_{\mathcal{O}_i}(x) \leq t\,(p_i)^2\} \subset \mathfrak{C}$ *with* $\mathbb{N}_{\mathcal{O}_i}(x)$ *the noise of the output ciphertext of* $\mathcal{O}_i$*. It can also be expressed as:* $\mathcal{S}_{\mathbb{N}}(\mathfrak{G}) = \bigcap_{i \in I} \mathcal{S}_{\mathbb{N}}(\mathcal{O}_i)$.

As we defined the feasible set for the noise, we can also define other feasible sets for other constraints, for instance to limit the size of the public keys, ciphertexts or even to add some constraints between parameters, etc. To be generic, we will refer to these additional feasible sets as one unique feasible set (intersection of all of these feasible sets) named $\mathcal{S}_{\mathsf{other}}(\mathfrak{G})$.

Formally what we want to compute is:

$$\underset{\mathfrak{G},x}{\arg\min}\, \mathsf{Cost}(\mathfrak{G}, x) \begin{cases} \mathfrak{G} \in \mathcal{S}_{\mathsf{FHE}}\left(\overline{\mathfrak{G}}\right) \\ x \in \mathcal{S}_{\mathsf{other}}(\mathfrak{G}) \bigcap \mathcal{S}_{\mathbb{N}}(\mathfrak{G}) \end{cases}$$

with $\mathcal{S}_{\mathsf{FHE}}\left(\overline{\mathfrak{G}}\right)$, all the possible plain graph translations, and $\mathcal{S}_{\mathsf{other}}(\mathfrak{G}) \bigcap \mathcal{S}_{\mathbb{N}}(\mathfrak{G})$, all the possible parameters satisfying the noise constraints as well as the other defined constraints.

Another essential notion in the optimization framework is the categories to sort any FHE operator according to the way they alter the noise. We describe them in the following definition.

Definition 11 (FHE Operator Categories) *We divide the FHE Operators into three categories regarding their respective noise formulae:*

(i) an operator which outputs a noise independent of the input noise, such as the PBS in our context;

(ii) an operator which adds to the input noise an independent noise, such as a KS or a modulus switching;

(iii) an operator which outputs ciphertexts with noise depending on the input noise (but not linearly), such as an homomorphic dot product or a tensor product.

Indeed, any FHE operator using the FFT does not exactly follow its theoretical noise formula because of the noise added by the floating point representation. In particular, simply by casting from a *u64* to a *f64* some of the LSB are lost. Similarly, the error grows all along computations due to the floating point arithmetic. To correct the formula accordingly, one solution is to collect data regarding the noise

in many different parameter settings and use them to deduce a corrective formula that takes into account the FFT-induced error.

Before trying to solve the optimization problem, we want to simplify it, otherwise it might take too much time to find the best solution. Only then, we can use the well known **branch-and-bound algorithm** to find the optimal parameters.

3.2 Pre-Optimization & Graph Transformations

Any sequences composed of ciphertext-ciphertext additions and ciphertext-plaintext additions/multiplication can be fused into a single homomorphic **dot product** (DP) between a vector of ciphertexts, with noise independent values, and a vector of plaintexts. The noise growth during a ciphertext-plaintext DP is simple to estimate as long as the noises in each ciphertext are independents from one another. In fact, the output variance of an homomorphic DP is the input variance multiplied by $\nu^2 = \sum_{i=1}^{\alpha} w_i^2$ where ν is the 2-norm of the plaintext vector $(w_1, \cdots, w_\alpha)$. In this paper, we will always group a sequence composed of the operators above-mentioned into an homomorphic DP whenever one happens in a graph.

We now introduce the notion of atomic patterns (AP) and their associated AP types. They will play an important role to boost the resolution of the optimization problem.

Definition 12 (Atomic Pattern Type) *An AP type $\mathscr{A}^{(\cdot)}$ corresponds to a sub-graph of FHE operators.*

An AP A is a particular instance of an AP type $\mathscr{A}^{(\cdot)}$. Two AP from the same type will correspond to the same FHE DAG, but they may have different parameters.

When we instantiate $A \in \mathscr{A}^{(\cdot)}$ with a parameter set x, we write $A(x)$. An important feature of $A(x)$, is that we can estimate its amount of noise at any edge of its FHE sub-graph and we can also estimate its total cost using the cost model.

Given a graph of homomorphic operators, we want to **transform** it into a graph of AP, i.e. splitting it into sub-graphs corresponding to some AP types. Then the noise feasible set $\mathcal{S}_{\mathbb{N}}$ becomes an intersection of the noise feasible sets of all the AP in the graph.

Some APs of the same type can be compared prior to the optimization and we can simplify the computation of $\mathcal{S}_{\mathbb{N}}$ by only keeping the feasible sets of the AP that are not included in others. We described the method on $\mathcal{S}_{\mathbb{N}}$ but the same can be done on $\mathcal{S}_{\mathsf{other}}$

We introduce the notion of **domination between AP**. An AP A dominates A' if any x satisfying the noise constraints (and the others) of A' also satisfies constraints of A. For all AP types, for all APs of this type, we will only keep the ones that are not dominated by any other AP. Indeed, we can discard the APs that are dominated because their constraints will be satisfied if the constraints of a dominant AP are satisfied.

Theorem 2 (AP Domination) *Let's consider $A_1, A_2 \in \mathbb{A}^{(\cdot)}$ two AP of a type that include an homomorphic DP, ν_1, ν_2 two 2-norms such that $\nu_1 \leq \nu_2$ and t_1, t_2 two noise bounds where $t_2 \leq t_1$. We have: $\mathbb{S}_{\mathbb{N}}(A_2(\nu_2, t_2)) \subset \mathbb{S}_{\mathbb{N}}(A_1(\nu_1, t_1))$.*

Proof 1 (Sketch) *A_1 and A_2 share the same type. When decreasing the noise bound, i.e. going from t_1 to t_2, we have less possible solutions x, but all the ones that satisfy t_2 will satisfy t_1. The same reasoning works for the 2-norms. By increasing the 2-norm, i.e. going from ν_1 to ν_2, there are less possible solutions x, but all the solutions satisfying ν_2 will satisfy ν_1.* □

It is then possible to build a graph of dependencies between AP. It is very useful for efficiently removing many APs and also find faster strategies to work with the APs that are not dominated.

3.3 Implemented AP Types

In this paper we will define a few AP types, explore relationships between them and compare them.

Definition 13 (Instantiated AP Types) *They all have an identifier and they correspond to different FHE sub-graphs:*

- *$A \in \mathbb{A}^{(1)}$ is composed of a DP, followed by a KS and a final PBS (i.e. a MS, a BR and a SE) as in [CJP21];*
- *$A \in \mathbb{A}^{(2)}$ is composed of a KS, followed by a DP and a final PBS (i.e. a MS, a BR and a SE) as in [CGGI20];*
- *$A \in \mathbb{A}^{(3)}$ is composed of a DP, followed by a KS and a final many-LUT PBS introduced in [CLOT21];*
- *$A \in \mathbb{A}^{(4)}$ is composed of a DP, followed by a KS and a final multi-value PBS introduced in [CIM19];*
- *$A \in \mathbb{A}^{(5)}$ is composed of a DP, followed by a KS and a final WoP-PBS which comes from Liu et al [LMP21]. This AP type does not require a bit of padding in plaintext encoding;*
- *$A \in \mathbb{A}^{(6)}$ has the exact same sequence of homomorphic operators than AP type #1 and the same cost. However, it is an imaginary AP type in the sense that we suppose that the FFT involved in the PBS does not add any noise, i.e. have an infinite precision, or at least enough to avoid it.*
- *$A \in \mathbb{A}^{(7)}$ is composed of a DP, followed by a KS and a final new WoP-PBS that we introduced in this paper (Section 4.1). This AP type does not require a bit of padding in plaintext encoding;*

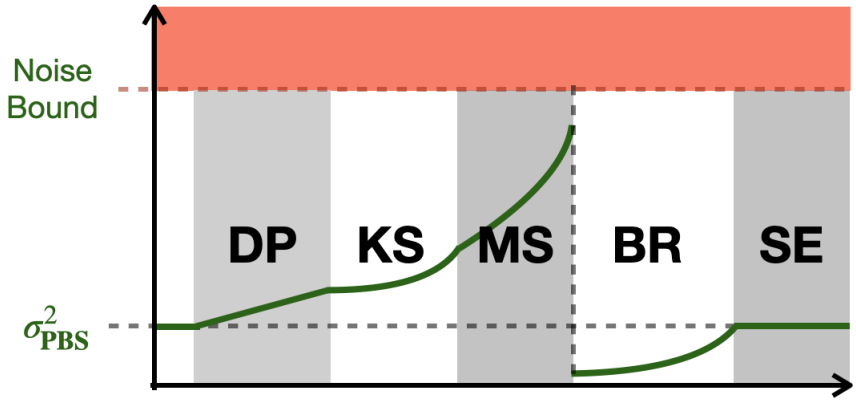

Figure 1: Noise evolution in an atomic pattern of type #1. The variance in a ciphertext after a PBS is noted $\sigma^2_{\mathbf{PBS}}$.

- $A \in \mathscr{A}^{(8)}$ *is composed of a DP, followed by a final PBS (i.e. a MS, a BR and a SE);*
- $A \in \mathscr{A}^{(10)}$ *is composed of a DP, a KS and the tree-PBS algorithm [GBA21];*

Figure 1 shows the noise evolution across an AP of type 1. Input ciphertexts of such an AP come with a minimal noise of variance $\sigma^2_{\mathbf{PBS}}$, i.e. the variance after the PBS. We see that during the DP, the KS and the MS, the noise keeps growing and when parameters are correctly picked, the amount does not cross over the noise bound. Then the BR operates over a bootstrapping key composed of fresh ciphertexts, to output a GLWE ciphertext with a noise variance of $\sigma^2_{\mathbf{PBS}}$. The SE does not change the amount of noise and output an LWE ciphertext.

As the noise grows with each operator inside the AP until reaching its peak after the Modulus Switching, it is enough to only check that the noise after this operator is below the noise bound.

Any FHE operators, except the ones of type (i), increase the noise inside the ciphertext. However, operators of type (iii) increase the noise depending on the input noise, hence the following theorem.

Theorem 3 (Relation Between $\mathscr{A}^{(1)}$ and $\mathscr{A}^{(2)}$) *We consider two 2-norms $\nu_1, \nu_2 \in \mathbb{R}^+$ such that $\nu_1 \leq \nu_2$, two noise bounds $t_1, t_2 \in \mathbb{N}$ such that $t_2 \leq t_1$ and two AP: $A_1 \in \mathscr{A}^{(1)}$ and $A_2 \in \mathscr{A}^{(2)}$. We have $\mathcal{S}_{\mathbb{N}}(A_2(\nu_2, t_2)) \subseteq \mathcal{S}_{\mathbb{N}}(A_1(\nu_1, t_1))$.*

Proof 2 *Let's start with the observation that $\mathscr{A}^{(1)}$ and $\mathscr{A}^{(2)}$ share the same search space $\mathscr{C}$ because they are built with the same operators. For a parameter set $x \in \mathscr{C}$, the maximum noise variance in an AP of type 1 is $\sigma^2_{\mathsf{out},1}(x) = \sigma^2_{\mathsf{in}}(x) \cdot \nu^2 + \sigma^2_{\mathbf{KS}}(x) + \sigma^2_{\mathbf{MS}}(x)$ where $\sigma^2_{\mathbf{KS}}(x)$ is the noise added by the KS and $\sigma^2_{\mathbf{MS}}(x)$ is the noise added by the MS. Similarly, the maximum noise variance in an AP of type 2 is $\sigma^2_{\mathsf{out},2}(x) = (\sigma^2_{\mathsf{in}}(x) + \sigma^2_{\mathbf{KS}}(x)) \cdot \nu^2 + \sigma^2_{\mathbf{MS}}(x)$.*

We consider $\bar{x} \in \mathcal{S}_{\mathbb{N}}(A_2(\nu_2, t_2))$, so we have $\sigma^2_2(\bar{x}) = (\sigma^2_{\mathsf{in}}(\bar{x}) + \sigma^2_{\mathbf{KS}}(\bar{x})) \cdot \nu_2^2$ and $\sigma^2_{\mathsf{out},2}(\bar{x}) < t_2^2$. The simplest non-trivial DP possible is when we only have one

input ciphertext multiplied by 1*, so we have* $1 \leq \nu$*. Since variances are positive and* $1 \leq \nu_1 \leq \nu_2$*, we have:*

$$t_1^2 \geq t_2^2 > \sigma_{\mathsf{out},2}^2(\bar{x}) = \sigma_{\mathsf{in}}^2(\bar{x}) \cdot \nu_2^2 + \sigma_{\mathbf{KS}}^2(\bar{x}) \cdot \nu_2^2 + \sigma_{\mathbf{MS}}^2(\bar{x})$$
$$\geq \sigma_{\mathsf{in}}^2(\bar{x}) \cdot \nu_2^2 + \sigma_{\mathbf{KS}}^2(\bar{x}) + \sigma_{\mathbf{MS}}^2(\bar{x}) \geq \sigma_{\mathsf{in}}^2(\bar{x}) \cdot \nu_1^2 + \sigma_{\mathbf{KS}}^2(\bar{x}) + \sigma_{\mathbf{MS}}^2(\bar{x}) = \sigma_{\mathsf{out},1}^2(\bar{x})$$

So we have $t_1^2 \geq \sigma_{\mathsf{out},1}^2(\bar{x})$*, meaning that* $\bar{x} \in \mathcal{S}_{\mathbb{N}}(A_1(\nu_1, t_1))$*, so* $\mathcal{S}_{\mathbb{N}}(A_2(\nu_2, t_2)) \subseteq \mathcal{S}_{\mathbb{N}}(A_1(\nu_1, t_1))$*.* □

This theorem is easily generalized to any couple of AP types where the difference is an order inversion between two operators of type (ii) and type (iii) and where the noise is monotonically non-decreasing (cf Section 5.4).

Dealing with several public keys increases a lot the complexity of the optimization problem, but may speed up the execution. The following theorems introduce a way to, prior to the optimization, characterize the optimal solutions, reducing the size of the search space.

Theorem 4 (Optimal KSK) *Let* KSK_0 *and* KSK_1 *two KS keys obtained through the resolution of the optimization problem. W.l.o.g., let us assume that a KS using* KSK_0 *adds strictly less noise than the one using* KSK_1*, then the KS with* KSK_0 *will be slower than the KS with* KSK_1*.*

Proof 3 *The two keys must have different parameters for the base and/or the number of level, because the noise added is different by hypothesis. If the optimization has selected two distinct keys it means that they both satisfy a different cost/noise trade-off. Then, if a KS with* KSK_1 *is slower than a KS with* KSK_0 *and generates more noise,* KSK_1 *will always be worse (both in terms of noise and cost) than* KSK_0 *which contradict the hypothesis that they are both optimal solutions.* □

Theorem 5 (Several KSK) *Let* $\mathfrak{G} = \{A(\nu_i, t)\}_{0 \leq i < Y}$ *an FHE graph only composed of AP type #1 such that* $\nu_0 < \nu_1 < \nu_2 < \cdots < \nu_Y$*. We consider that we can have several possible KSK. The optimal* $\vec{\delta} = (\delta_0, \cdots, \delta_{Y-1})$ *has the property that for all* $0 \leq i < Y - 1$ *there is* $\delta_i \geq \delta_{i+1}$*.*

Proof 4 (Sketch) *Following the same logic as in the toy example above, it is easy to prove this theorem.* □

Another interesting approach in the optimization, is to automatically insert PBS during a DP operator wherever it is interesting with regards to the cost model. Indeed, as the PBS is the most costly operator, inserting PBS can increase the total cost of the computation. But on the other hand, if the 2-norm of a DP operator is high, the parameters must be large enough to still guarantee the correctness of the computation. Those large parameters will have an impact on the cost and so we need our framework to choose whether it is interesting to split the DP operator or not.

The following theorem explores this approach.

Theorem 6 (DP Splitting) *Let $A(\nu, t)$ be an AP of noise bound t and including a DP of 2-norm ν. Let $\widetilde{A}(\nu, t, d)$ the same AP than A but where its DP is split into $d+1$ sub-DP of approximately the same 2-norm and connected together with PBS. It actually breaks an AP into $d+1$ AP of the same type organised in two layers (d followed by a last one connecting them all).*

Let $\mathfrak{G} = \{A(\nu_i, t)\}_{0 \leq i < Y}$ such that $\nu_0 < \nu_1 < ... < \nu_{Y-1}$. Let $\vec{d^*} = (d_0^*, \cdots, d_{Y-1}^*)$ and $\vec{d} = \left(d'_0, \cdots, d'_{Y-1}\right)$ be two possible splitting solutions. We define the following two FHE graphs: $\mathfrak{G}^* = \left\{\widetilde{A}(\nu_i, t, d_i^*)\right\}_{0 \leq i < Y}$ and $\mathfrak{G}' = \left\{\widetilde{A}(\nu_i, t, d'_i)\right\}_{0 \leq i < Y}$.

If every coordinates of d^* is inferior or equal (coordinate wise) to the ones from $\widetilde{d}$ and $\mathbb{S}_{\mathbb{N}}(\mathfrak{G}^*) = \mathbb{S}_{\mathbb{N}}\left(\widetilde{\mathfrak{G}}\right)$, then, $\widetilde{d}$ cannot be the optimal solution.

Proof 5 (Sketch) *As before, we use the fact that the noise feasible set of those two DAG is the same and the cost of one DAG is higher than the other as it contains more PBS.* □

We came up with a few AP types, but many others could be studied. We believe that this framework could include other GLWE-based schemes and that we start a generic and versatile approach to automatically find cryptographic parameters.

4 Homomorphic Integers

In this section we introduce a new procedure to compute a WoP-PBS, i.e. a PBS where the input ciphertext does not need a bit of padding in the MSB of the plaintext. Then, we propose new ways to encrypt and work on large modular integers. Those constructions can be based on traditional TFHE PBS or on our new WoP-PBS or even other variants of the PBS. We conclude this section by explaining how to compute LUT evaluations on the large modular integers we described.

4.1 New WoP-PBS

The WoP-PBS, i.e., a PBS which does not require a bit of padding, was introduced for the first time by Chillotti et al. [CLOT21]. It takes as input an LWE ciphertext with or without bits of padding in the MSB, a public key called bootstrapping key, and a LUT L. It outputs the homomorphic evaluation of the LUT on the input message, i.e. an LWE encryption of $L[m]$. At least one bit of padding is required in the original TFHE-like PBS algorithms for an encrypted message m that is not a bit.

We improve the state of the art when it comes to WoP-PBS. Indeed, our new algorithm is able to take as input not only one LWE ciphertext but several, and the complexity scales well in the number of LWE ciphertexts. It is also able to round (or truncate or more) each of the input messages to a given precision. Finally, it can be used to compute several LUT on the same set of inputs at the cost of (about) a single LUT.

Our method is based on two building blocks: the circuit bootstrapping and the mixed packing from [CGGI20]. We provide the details of the technique (using vertical packing in this case) in Algorithm 1.

Remark 1 (Circuit Bootstrapping Optimizations) *The second step of the circuit bootstrapping which is the packing functional key switchings can be improved by following a similar footstep as a technique proposed in [CCR19]. We perform an initial LWE-to-GLWE KS (not functional) to each of the outputs of the PBS, and then, as already done in [CCR19], we perform an external product times the GGSW encryption of the GLWE secret key to obtain the remaining GLWE ciphertexts. This allows us to reduce the size of public evaluation keys. Furthermore, increasing the GLWE dimension k allows to reduce the degree N of the GLWE polynomials. One can keep the same security level and, at the same time, work with smaller polynomials, making the FFT faster. Obviously, reducing N reduces the precision of the PBS, but we only bootstrap one bit in a circuit bootstrapping here.*

Remark 2 (Algorithm 1 Optimizations) *The* KS-PBS *performed in Line 5 of Algorithm 1 is a Generalized PBS, as described in [CLOT21], so the modulus switching directly reads the next bit to be extracted. The sign function is evaluated and in order to re-scaled the bit at the right scaling factor.*

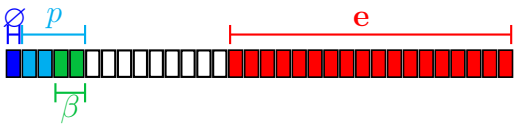

Figure 2: Plaintext binary representation with a base $\beta = 4 = 2^2$ (green), a carry subspace (cyan), and a carry-message modulo $p = 16 = 2^{2+2}$ (cyan+green) such that $0 < \beta < p$. A bit of padding is displayed in the MSB (dark blue). So the plaintext modulo is $32 = 2^{2+2+1}$. This means that we have 2 bits in the carry subspace (set to 0 in a fresh ciphertext), that will contain useful data when one computes leveled operations.

The circuit bootstrappings used in Lines 8 and 9 are also instantiated with a Generalized PBS. If we chose a value of $\vartheta > 0$ we could improve the circuit bootstrappings with a PBSmanyLUT, as described in [CLOT21].

We can also observe that one of the PBS of the circuit bootstrappings used in Line 8 could be avoided thanks to the KS-PBS *in Line 5, that might already provide the bit extracted at the right re-scaling factor.*

4.2 Advanced Modular Arithmetic with a Single LWE Ciphertext

In this section we describe an advanced mode for evaluating operations on modular integers. It is built in a parametrizable manner enabling many optimizations. The core idea is to allow room in a ciphertext encrypting integers modulo $\beta \in \mathbb{N}$ to store big enough carries coming from leveled operations. We introduce a metadata keeping track of the worst case message in each ciphertext to guarantee correctness of all the described algorithms and optimize the computation.

Plaintext Space Format We describe here a framework to compute homomorphic operations on modular integers. Let $\beta \in \mathbb{Z}$ be the previous mentioned modulus. We split the traditional plaintext space into three different parts: the *message subspace* storing an integer modulo β (we call β the base), the *carry subspace* containing information overlapping β, and a bit of padding (or more) often needed for bootstrapping. In this context, we refer to the *carry-message modulo* as the subspace including both the message subspace plus the carry subspace, and we note it $p \in \mathbb{N}$. Figure 2 shows a visual example.

We introduce a metric called *degree of fullness* that is associated to an LWE ciphertext and represents the largest message that this ciphertext can contain.

Definition 14 (Degree of Fullness) *Let $q \in \mathbb{Z}$ be the ciphertext modulus, let $p \in \mathbb{Z}$ be the carry-message modulus such that $p < q$. Let* ct *be a LWE ciphertext encrypting a message $0 \leq m < p$. Let μ be the known worst case for m, i.e., the biggest integer that m can be. We have $0 \leq m \leq \mu < p$ and we consider that there either one bit of padding or none: $\pi \in \{0, 1\}$.*

We define the degree of fullness of $ct \in \mathsf{LWE}_{\vec{s}}(\tilde{m})$ *with* $\tilde{m} = \mathsf{Encode}(m, 2^\pi \cdot p, q)$ *as the following number:* $\deg(\mathsf{ct}) = \frac{\mu}{p-1} \in \mathbb{Q}$. *To ensure correctness, the degree of fullness should always be a quantity included between* 0 *and* 1, *where* $\deg(\mathsf{ct}) = 1$ *means that the carry-message subspace is full in the worst case. It is required when using a traditional PBS or when the amount overlapping* p *is needed to properly decode.*

We take advantage of the carry subspace to compute leveled operations and to avoid bootstrapping. In practice, the carry subspace acts as a buffer to contain the carry information derived from homomorphic operations and the degree of fullness acts as a measure that reveals when the buffer cannot support additional operations: once this limit is reached the carry subspace is emptied by bootstrapping. To be able to perform a leveled operation between two LWE ciphertexts of that type, they need to have the same base β, carry-message p and ciphertext modulus q. Addition and multiplication by small constants can be performed as usual in TFHE. The computation of the opposite requires a correction term and the subtraction is a combination of opposite and addition operations. Univariate functions can still be computed with PBS.

We provide more details on these operations in Supplementary Material A, and we also provide details on how to build a LUT and evaluate the PBS on ciphertexts having p different from a power of 2.

Thanks to the carry buffer, when they are empty enough, we can evaluate multivariate functions. 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 and add it to ct_2 and finally compute a PBS on the concatenation. We multiply ct_1 by $\mu_2 + 1$, where μ_2 is the worst possible value that can be reached by the m_2. These strategy is possible if the obtained ciphertext (before PBS) respect the conditions on the degrees and on the noise. Once the two messages are concatenated in a single ciphertext, the bivariate LUT L can be simply evaluated as a univariate LUT L' on the concatenation of m_1 and m_2. A visual example is proposed in Figure 13 in Supplementary Material H. This concept can be easily extended to multivariate functions.

We use this approach to compute homomorphic multiplication, both in the LSB (i.e. $m_1 \cdot m_2 \mod \beta$) and in the MSB (i.e. $\left\lfloor \frac{m_1 \cdot m_2}{\beta} \right\rfloor$). If instead we want to compute the multiplication without any modular reduction, we can use well known techniques in TFHE literature such as in [CJL+20].

The last operations we need to describe enable to extract the carry or the message when the degree of fullness has reached a limit (or whenever one needs to extract them). We call these operations carry and message extract: they both are performed as a PBS. We provide details on multiplications and on carry and message extractions in Supplementary Material A.

Generally with FHE one has to monitor noise growth. However, in this paper,

we chose 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. We give more details in Section 5.1.

4.3 Radix-Based Large Integers

We extend the approach proposed in previous section to larger messages encrypted as multiple ciphertexts. We introduce a first generic method to perform homomorphic computations modulo a larger integer $\Omega \in \mathbb{N}$.

Any homomorphic modular integer we describe here is a list of $\kappa \in \mathbb{N}$ blocks (LWE ciphertexts), along with a list of radix-bases β_i with $0 \leq i < \kappa$, and we call them *radix-based modular integers*. With such a list of ciphertexts, we are able to represent a large integer modulo $\Omega = \prod_{i=0}^{\kappa-1} \beta_i$. For each block, we actually need a couple $(\beta_i, p_i) \in \mathbb{N}^2$ of parameters to be defined, which respectively corresponds to the message subspace and the carry-message subspace involved with the modular arithmetic described in Section 4.2. In Figure 12 in Supplementary Material H, we give a visual representation out of a toy example of a radix-based modular integer.

In practice, the restriction for Ω is that it has to be a product of small basis. Indeed, TFHE-like schemes do no scale well when one is increasing the precision, so the good practice is to keep $p_i \leq 2^8$. We describe later a more refined procedure enabling the use of any $\Omega \in \mathbb{N}$.

To encode a message $\mathsf{msg} \in \mathbb{Z}_\Omega$ and encrypt it as a radix-based modular integer, one needs to know the list of integer parameters $\{\beta_i, p_i\}_{0 \leq i < \kappa}$ such that $\Omega = \prod_{i=0}^{\kappa-1} \beta_i$, and for all $0 \leq i < \kappa$ we have $2 \leq \beta_i \leq p_i$. The first step is to decompose msg into a list of $\{m_i\}_{i=0}^{\kappa-1}$ such that $\mathsf{msg} = m_0 + \sum_{i=1}^{\kappa-1} m_i \cdot \left(\prod_{j=0}^{i-1} \beta_i\right)$. Then we can independently call the Encode function (Definition 2) on each m_i so we have $\tilde{m}_i = \mathsf{Encode}\,(m_i, 2^\pi \cdot p_i, q)$ with π the number of bit of padding. Finally we can encrypt each $\tilde{m}_i$ into an LWE ciphertext. To decode, we simply recompose the integer from the m_i values.

Arithmetic Operations We can now describe the operators that we are able to compute over radix-based modular integers. Note that those operators are allowed as long as both radix-based modular integers are encoded and encrypted with the same parameters $\{(\beta_i, p_i)\}_{0 \leq i < \kappa} \in \mathbb{N}^{2\kappa}$ and ciphertext modulus q.

For the computation of addition we use a schoolbook approach, and keep everything correct thanks to the degree of fullness storing the worst message for each block. To compute the negation, we start by negating the least significant block and by adding a correction term. Then, we repeat the operation for the following blocks. We provide a toy example in Supplementary Material B. To compute the subtraction we simply combine an opposite computation with an addition. To compute the multiplication we use a schoolbook approach, as for addition. We provide a toy example to better understand in Supplementary Material B. To compute a multiplication between a ciphertext and a constant $c \in \mathbb{Z}_\Omega$, we use a similar approach

as the schoolbook multiplication, combining PBS and leveled additions. There are several approaches that can be used to compute a generic LUT over a radix-based modular integer. We describe them in Section 4.5.

Buffers for carries are limited in each block. This is why we often need to make room. To keep the encoding coherent, we have to propagate the carry of a block into the next one.

A simple method is to compute both the carry extraction and the message extraction for the least significant block, and then adding the extracted carry to the second block, and keep doing it until emptying all the carry buffers. Note that the most significant block's carry buffer can be thrown away, i.e. only computing the message extraction for this block is required.

Generalization to Any Ω Until now we have used a modular integer Ω that is equal to the product of the bases, i.e., $\Omega = \prod_{i=0}^{\kappa-1} \beta_i$. In this section we take away the restriction on Ω and we generalize the techniques presented in this section to a more generic base $\prod_{j=0}^{\kappa-2} \beta_j < \Omega < \prod_{j=0}^{\kappa-1} \beta_j$.

All the algorithms stay the same, with the exception of the modular reduction which is trickier to compute. We propose *two methods* to perform this modular reduction. There is no best method though, both of their efficiency depend on Ω and the product of the basis selected.

We propose a *first method* consisting in performing multiple LUT evaluations in the most significant block to reduce it modulo Ω. Suppose we have κ blocks as input, we take the MSB block $\mathsf{ct}_{\kappa-1}$ which is supposed to have an almost full degree (otherwise one does not need to call this procedure yet). Remember that this block stores a "small" value $m_{\kappa-1} < p_{\kappa-1}$ but it represents a "big" value with this encoding: $m_{\kappa-1} \cdot \prod_{i=0}^{\kappa-2} \beta_i$. We generate κ new blocks to store its modular reduction in the defined radix-bases. The final step is to add this to the $k-1$ left input blocks to end up with the desired result. We provide a detailed description of the method in Algorithm 4 in Supplementary Material D.

We propose a *second method* which is more convenient when Ω and the bases $(\beta_0, \ldots, \beta_{\kappa-1})$ respect a certain condition. The idea is based on the shape of $-\prod_{h=0}^{\kappa-2} \beta_h$ reduced modulo Ω. We look at its radix decomposition:

$$-\prod_{h=0}^{\kappa-2} \beta_h \mod \Omega = \nu_0 + \nu_1 \cdot \beta_0 + \nu_2 \cdot \beta_0\beta_1 + \ldots + \nu_{\kappa-1} \cdot \prod_{j=0}^{\kappa-2} \beta_j.$$

If $\nu_{\kappa-1} = 0$ and the other elements of the decomposition, i.e. $\nu_0, \nu_1, \ldots, \nu_{\kappa-2}$, are small integers (ideally many of them set to 0), then this method is very useful.

Indeed, when these conditions are respected, the idea is to replace the MSB block by multiplying it to the non-zero constants ν_j and subtracting the results to the j-th input block. Some multiplications with positive constant are needed and might require some carry propagation prior to them depending on the degrees of fullness. This method is detailed in Algorithm 5 in Supplementary Material D.

4.4 CRT-Based Large Integers

In this section we propose a new way to build encryption of large modular integers based on TFHE-like schemes and exploiting the CRT. In FHE schemes, the CRT is often used to speed up computations on the integers composing a ciphertext. Here we use the CRT at the encoding level, i.e. before encryption.

We introduce a hybrid approach that include radix-based decomposition inside the CRT representation, to overcome some of the limitations inherent to the CRT-only approach.

The idea is to use several LWE ciphertexts to encrypt sub-messages, modulo the considered CRT residues. To encode a message $\mathsf{msg} \in \mathbb{Z}_\Omega$ and encrypt it as a CRT-based modular integer, one needs to know the list of integer parameters $\{\beta_i, p_i\}_{0 \leq i < \kappa}$ such that $\Omega = \prod_{i=0}^{\kappa-1} \beta_i$, for all $0 \leq i < \kappa$ we have $2 \leq \beta_i < p_i$ and because of the CRT we need each couple β_i and $\beta_{j \neq i}$ of bases to be co-primes. The first step is to compute the CRT on $\mathsf{msg} \in \mathbb{Z}_\Omega$ so it is split into a list of $\{m_i\}_{i=0}^{\kappa-1}$ such that $\mathsf{msg} = m_i \mod \beta_i$ for all $0 \leq i < \kappa$, then to encode and encrypt. To decode, we simply need to compute the modular reduction modulo the bases β_i and compute the inverse of the CRT.

With this CRT encoding, we have to empty the carry buffers when they are (almost) full. Indeed, the quantity overlapping the base β_i is not needed to maintain correctness but when using TFHE PBS, the bit of padding needs to be preserved. We need to only call the message extraction algorithm, described in Section 4.2 when needed.

Arithmetic operations such as additions, subtractions, multiplication between two encrypted integers, or between an encrypted integer and a known constant, or even computing an opposite can be easily done with data encoded with a CRT. We can use all the operators described in Section 4.2.

Homomorphic LUT evaluation is possible with some heavy computation taking into account the entire encrypted message (see Section 4.5). However, if the LUT is more friendly to our CRT encoding, one can compute it faster. For instance, the best case scenario is when we can describe L as $\mathsf{Encode}_{\mathsf{CRT}}\left(L\left(\mathsf{msg}\right)\right) = \left(L_0\left(m_0\right), \cdots, L_{\kappa-1}\left(m_{\kappa-1}\right)\right)$ where $\mathsf{Encode}_{\mathsf{CRT}} = (m_0, \cdots, m_{\kappa-1})$. This kind of LUT only requires one LUT evaluation per sub-message m_i. We can imagine less friendly functions but can still be sped up in the same manner, $\mathsf{Encode}_{\mathsf{CRT}}\left(L\left(\mathsf{msg}\right)\right) = \left(L_0\left(m_0\right), \cdots, L_\alpha\left(m_\alpha\right), L'\left(m_{\alpha+1}, \cdots, m_{\kappa-1}\right)\right)$ where $\mathsf{Encode}_{\mathsf{CRT}} = (m_0, \cdots, m_{\kappa-1})$, where some of the computations can be done independently and the computed with the generic approach.

Advantages & Limitations As explained above, with the CRT encoding, there is no need to propagate carries to the next block, avoiding a lot of computation and make the computation faster. We actually exploit the room we have in the carry buffer in each block to have more leveled operations computed between two message extractions (to empty the carry). Another advantage of this encoding is that it

enables a parallel execution of the operators at the block level, whereas with the radix approach, the carry propagation or carry reduction mostly implies sequential operations.

This CRT encoding is very efficient but suffers from the CRT requirements, i.e. co-prime bases, and the precision limitation we have in practice with TFHE. Indeed, there are a limited number of primes between 2 and 128. It means that this approach is good when Ω is composed of small enough co-prime factors but for the rest of the possible Ω we need other solutions.

Remark 3 (Fast & Native CRT Implementation) *We implemented a fast version of CRT encoded integers where additions and scalar multiplications are native with 16 bits of precision. Indeed, they are extremely fast since they do not need any PBS to be involved. We used the CRT basis* $(\beta_0 = 7, \beta_1 = 8, \beta_2 = 9, \beta_3 = 11, \beta_4 = 13)$ *and we used for each odd residue a non power-of-2 encoding. For example, to encode* $m_0 = 3 \mod 7$*, the plaintext is* $\left\lfloor 3\frac{q}{7} \right\rceil$*. Note that there is no bit of padding, and no need to use a degree of fullness either. To compute additions we use the LWE addition on each residue, and to compute a scalar multiplication by* α*, we decompose* α *with our CRT basis into smaller integers, and compute scalar multiplications with them. To evaluate a LUT or compute a multiplication we use the new WoP-PBS. We simply need to extract the first* $\lceil \log_2(\beta_i) \rceil$ *most significant bits for odd* β_i *(from LSB to MSB). The first bit extraction has to be computed differently here: instead of shifting of* $\frac{q}{4}$ *we shift of* $\frac{q}{64}$*. We generated with the optimization framework parameters for DP of 2-norm of 8 bits (for each residue computed out of the decomposition of the scalars), which allows quite a lot of leveled operations. Many optimizations remain possible or better CRT basis could be used, but we give preliminary benchmarks in Supplementary Material in Table 1. Note that this approach is very fast because of the "free" leveled computation and involves less PBS than the radix approach, for instance, which leads to smaller failure probabilities for an entire graph.*

Remark 4 (Hybrid Approach) *To overcome the above-mentioned limitations, we propose a new homomorphic hybrid representation: the idea is to use the CRT approach (from Section 4.4), and to represent the larger CRT residues by using radix-based modular integers (from Section 4.3) when needed. In practice, with this new hybrid approach, we do not have any restriction on* Ω*. We provide more details in Supplementary Material C.*

4.5 Discrete Function Evaluation

In this section we describe a new method to homomorphically evaluate a discrete function over data encrypted in the various ways described in this paper. The PBS takes as input a single LWE ciphertext and it is able to evaluate the LUT on the encrypted message. However, when the message is encoded in multiple LWE ciphertexts, a single PBS is not enough.

Two techniques could be used to evaluate a LUT over a large integer. *The first one* is the TreePBS method proposed in 2021 by Guimarães, Borin and Aranha [GBA21], which enables to evaluate a large look-up table over many input ciphertexts. Since this technique is not new, we provide details about how to use it for our large homomorphic integers in Supplementary Material D.1.

The second technique we use relies on the new WoP-PBS, introduced in Section 4.1 to evaluate some univariate (or multivariate) functions on radix-based, CRT-based and hybrid modular integer ciphertexts: the idea is to extract all the bits from one or more of the modular integers, up to their degree of fullness, and then evaluate the desired LUTs. Because we might have non-empty carry buffers, different possible inputs encode the same value. Hence the LUT L needs to contain some kind of redundancy. If the goal is to compute the discrete function f, one needs to compute the L as $L[(m_0, \cdots, m_{\kappa-1})] = \mathsf{Encode}\left(f\left(\mathsf{Decode}\left(m_0, \cdots, m_{\kappa-1}\right)\right)\right)$.

This approach is also very convenient for particular LUT such as the ReLU function in the radix mode. Indeed, we only need to use the MSB, so the CMux tree (in the WoP-PBS) is greatly simplified and becomes linear in the number of blocks. It is even better for the sign function, we can settle for CMux in the Cmux tree.

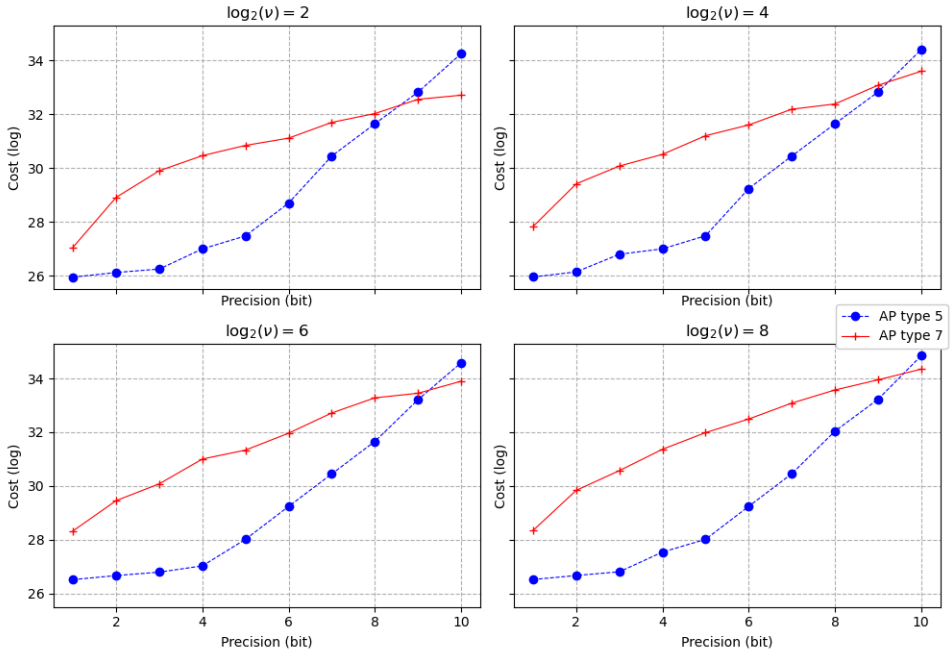

Figure 3: In this figure, we compare the cost of AP type #7 and type #5. The first one corresponds to DP-KS followed by our new WoP-PBS (Section 4.1), and the second one to DP-KS followed by the WoP-PBS from [LMP21].

5 Applications of Our Optimization Framework

One can address many FHE use-cases with TFHE. For instance to homomorphically compute the inference of some neural network [CJP21], only a few FHE operators are required: ciphertext/ciphertext and ciphertext/plaintext additions, ciphertext/-plaintext multiplications, KS and PBS. It can be represented with AP of type #1 or AP of type #7.

In this section we instantiate our optimization framework in different ways so we can find the best FHE parameters for the defined AP types. For instance, we look for the best parameters for TFHE when messages are not Boolean and for many different precisions or DP's 2-norms. This will enable fast computation with the new integer types introduced in Section 4. We will also be able for the first time, to fairly compare different FHE strategies and algorithms for bootstrapping and/or LUT evaluation and many other useful applications.

As explained in Section 3.2, we have to provide for each plain operator a list of FHE sub-graphs computing it leading to Guarantee (iii). To simplify the optimization we only provide a single sub-graph for each plain operator, i.e. $\left|\mathcal{S}_{\mathsf{FHE}}\left(\overline{\mathcal{G}}\right)\right| = 1$.

With all the AP types that we define in this paper, we make the assumption that the cost of a DP is negligible compared to the other FHE operators.

We also assume that the inputs of an AP are some outputs of another PBS. This assumption enables to compare different bootstrapping algorithms that have a different impact on the noise as they both need to satisfy the same correctness.

Regarding the confidence interval for the noise, we will use the same value $z^*(p_{err}) = 4$ in all the following applications. It means that each PBS has a failure probability of at most $2^{-13.9}$. We added a corrective term to the FFT-less noise formula given in [CLOT21] which is custom for FFTW [Pad11].

5.1 Optimal Parameters For Exact Non-Boolean TFHE

In this application, we demonstrate that we can use our optimization framework to find optimal parameters in different precision/2-norm contexts. We start with the advanced modular arithmetic use-case, explained in Section 4.2. We can find parameters whether we use TFHE's PBS or the WoP-PBS introduced in the paper, we will however focus on AP type #1 in this application.

We also make the following important hypothesis, once again to speed the optimization up, but also to limit the size of the public material needed to run the FHE DAG. We restrict ourselves to having up to one public key per FHE operators. This means that the cryptographic parameters for each AP will be the same, so the only differences between APs $A_i(\nu_i, t_i)$ of type #1 is their 2-norm ν, and their noise bound $t(p)$ only.

Using Theorem 2, we are able to remove many APs. Then, instead of having to check the constraints of as many feasible sets as we have atomic patterns in $\mathfrak{G}$, we only need to check the constraints of the remaining APs. After this simplification, there is at most as many AP as there is different noise bounds. With our selection of TFHE operators, it means 8 APs maximum.

As we neglected the cost of the DP, the only parameters impacting the total cost of an AP are the cryptographic parameters. So instead of having to compute the cost of the whole graph, we can settle for the cost of one AP.

We provide an extensive list of parameters for AP type #1 in Section I.3 in Supplementary Material, for precision up to 8 bits and their respective highest possible 2-norms. Using the message modulus p, the ciphertext modulus $q = 2^{64}$, $z^*\left(p_{\mathsf{err}}\right) = 4$ i.e. the probability of error $p_{\mathsf{err}} \approx 2^{-13.9}$ and one bit of padding (i.e. $\pi = 1$), we can compute the noise bound (definition 8) as $t\left(p, \pi = 1\right) = \frac{q}{2^{\pi+1} \cdot p \cdot z^*(p_{\mathsf{err}})}$.

5.2 WoP-PBS Comparison

A few WoP-PBS constructions have been proposed in the literature. Some works [KS21, LMP21] tried to compare them. Our optimization framework enables to truly compare such techniques together. We will work with AP of type #7 ending with the WoP-PBS introduced in this paper, and AP of type #5 ending with the WoP-PBS introduced by Liu *et al* [LMP21].

In Figure 3, we plot the results we obtained: each point correspond to an optimal parameter set found for a given AP type, precision and 2-norm.

Note that AP type #5 has to bootstrap the entire message, and above 8 bits of precision polynomials bigger than 2^{14} are needed. To fairly compare, we had to increase this limit even though the library could not support such big polynomials.

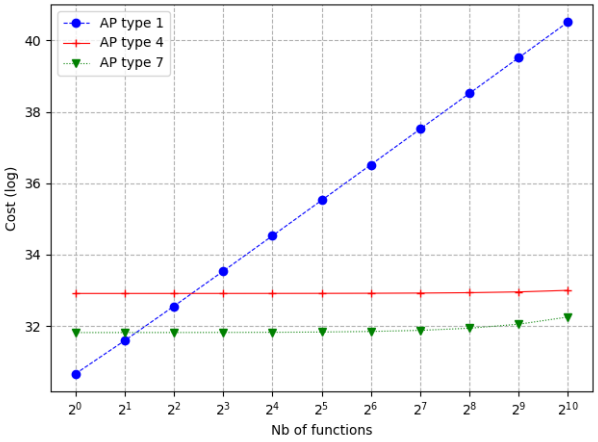

Figure 4: This figure compares the cost of AP type #7, #1, and #4. The first one corresponds to DP-KS followed by our new WoP-PBS, the second one to DP-KS-PBS, and the last one to DP-KS followed by a multi-value PBS introduced in [CIM19]. Precision is fixed to 8 bits and the 2-norm to 6 bits.

As we can see, the WoP-PBS introduced in this paper is faster with precisions above 8-bits, making it the best known way to deal with bigger precisions in TFHE-like schemes. When one has to work with smaller precision, they want to use the algorithm from [LMP21].

5.3 MISD-PBS Comparison

In this application we compare the Multiple Instructions, Single Data (MISD) operators that we have at our disposal for homomorphic LUT evaluation.

We compare three techniques: the naive one (baseline) where one computes α PBSs on the same input to get the α desired evaluations, a second one using the WoP-PBS introduced in this paper (Section 4.1), and a last one using the multi-value PBS introduced in [CIM19].

Figure 4 plots the cost of the optimal found parameters for the three different AP types and different number of functions to evaluate. We can see that both the WoP-PBS and the multi-value PBS scale well in the number of LUT to evaluate over the same input.

This figure does not mean that the WoP-PBS is better than the multi-value, it is indeed better in some particular settings like this one, with a precision of 8 bits and a 2-norm of 6 bits. Figure 10 in Supplementary Material, shows this same experiment in a setting where the multi-value is better.

From this experiment, we concluded that it is better to use the WoP-PBS for precisions above 8 bits and with high 2-norms.

5.4 KS Before or After PBS? No KS at All?

In TFHE [CGGI20], the KS is generally computed right after the PBS (type #2), as instance in gate bootstrapping, and not the other way around as in the AP type #1. This application aims to compare the two approaches and to answer which one is better. It also investigates what would happen if we were to remove entirely the KS with the AP type #8. To simplify, we fixed the GLWE dimension k to 1, i.e., we use RLWE ciphertexts.

Figure 5 plots the cost of the optimal parameter sets for the three AP types we mentioned, for an increasing precision and for several 2-norms of the DP. Sometimes there exists no parameter sets. We see that without any KS, it's impossible to find parameters when one increases the 2-norm or the precision. We also see that having the KS right before the PBS is more efficient than having it right after, as predicted by the theory. To conclude, one wants to always compute the KS right before the PBS as in AP #1.

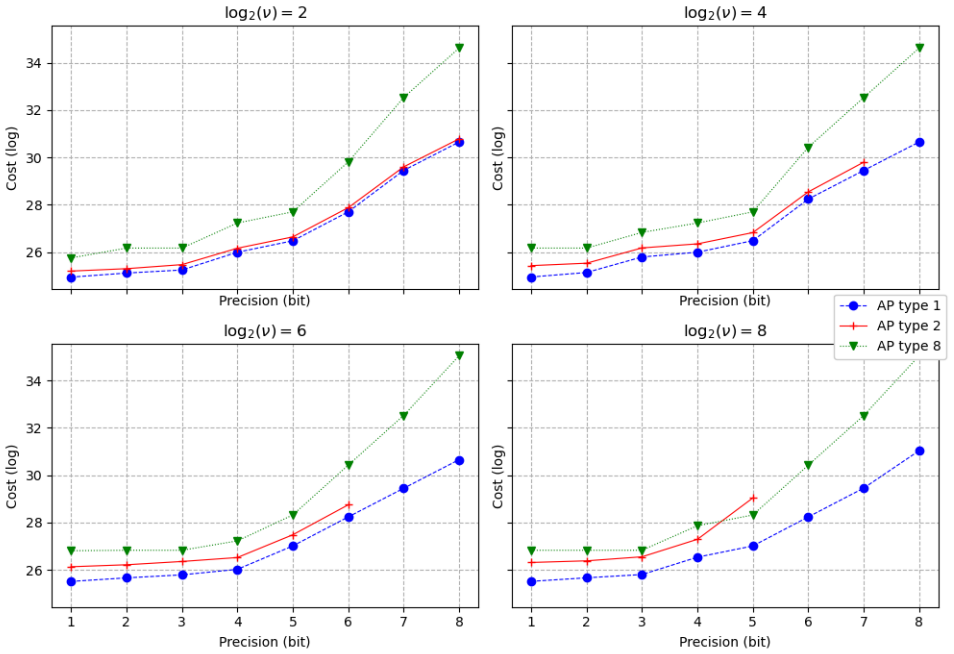

Figure 5: In this figure, we compare AP of type #1, type #2 and type #8. The first one corresponds to DP-KS-PBS, the second one to KS-DP-PBS and the last one to DP-PBS.

5.5 Multi-Input LUT Evaluation

In this application, we investigate the scalability of algorithms dedicated to the LUT evaluation over an input message split among several LWE input ciphertexts. We have implemented two algorithms, one base on our WoP-PBS, described in

Section 4.1, and the TreePBS algorithm [GBA21]. The two AP types that we consider are #7 and #10.

We plot in figure 6 the comparison with two input ciphertexts. Each point corresponds to the best possible parameter sets found with the optimization framework according to the context. We can see that when the precision increases, the TreePBS does not scale well and eventually no parameters can be found anymore (on this figure we allow polynomial sizes to be above the one supported in practice, i.e. 2^{14} for the TreePBS).

We have in figures 8 and 9 in Supplementary Material, the same experiments but for 3 and 4 LWE ciphertext inputs. We observed that when the number of input increases, once again the TreePBS approach does no scale well and it becomes even harder to find parameters.

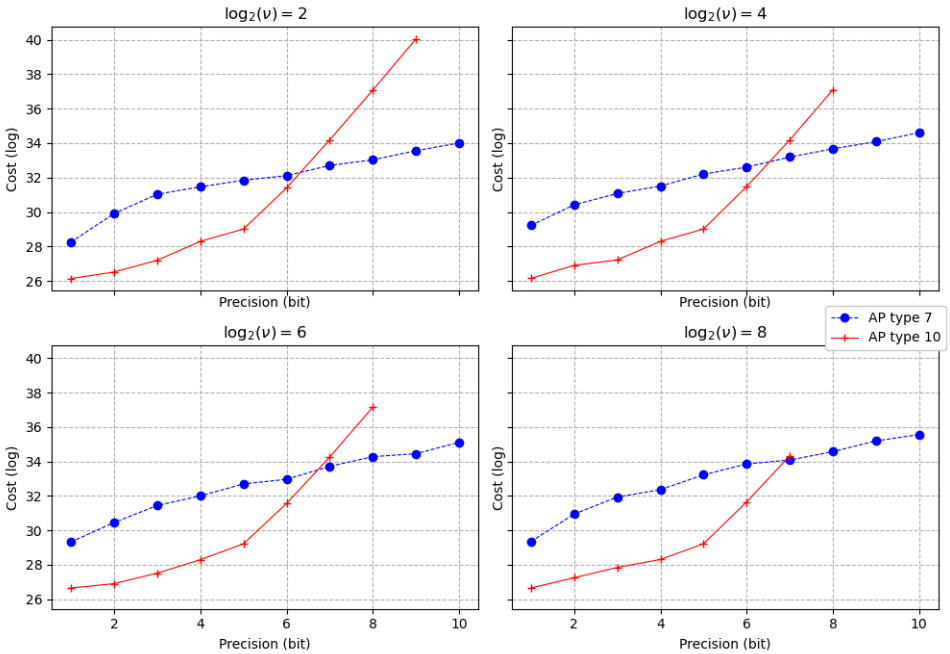

Figure 6: In this figure, we evaluate a LUT over 2 encrypted inputs, with on the one hand the WoP-PBS introduced in this paper, AP type #7 and on the other hand with the TreePBS [GBA21], AP type #10.

5.6 Other Applications

In this section we describe a few more applications offered by the optimization framework. A simple yet useful is the capability of exploring the correctness/efficiency spectrum. Indeed, the framework takes as input the LUT evaluation's probability of error so it is easy to go from more efficient parameters with a greater error probability to less efficient ones with a smaller error probability.

Consensus-Friendly TFHE & Blockchain Application Two implementations of the same FHE algorithm that does not involve the FFT will output the same result as long as it operates over the same inputs (same ciphertexts and same public materials). For instance, different implementation of a DP or an LWE-to-LWE KS will produce the same outputs.

However, implementations that leverage the FFT output different ciphertexts depending on the FFT algorithm involved. To highlight this, we made an experiment with the traditional parameter set of TFHE-lib for the bootstrapping. We use the same secret keys, the same bootstrapping key and the same input ciphertexts, but two different implementations of the PBS with their respective FFT implementations.

We computed the difference on the resulting ciphertexts for the two different implementations, we call this value the *error* of the ciphertexts, which is different from the noise needed for security in the plaintext. We observed that the ciphertexts had the same most significant bits but their least significant bits were different. We also re-run the experiment with different parameters: more levels and bigger polynomials in the bootstrapping key. The messages encrypted were still correct but the ciphertexts were completely different.

From those experiments, we can conclude that for a given parameter set and a ciphertext with a given input error, the PBS with a given FFT either resets the error to a minimal level or outputs the maximum amount of error, i.e., a re-randomization of the ciphertext. It means that an FHE circuit containing a DP, a KS and a PBS will not output the exact same ciphertext if it is run on the same inputs with different implementations. This is not compatible with use-cases where one actually needs to guarantee reproducibility across different implementations.

Thankfully, it is possible to ensure the reproducibility by tweaking a bit our optimization framework as well as the PBS algorithm. The idea is to use a new AP type that is identical to type #1 but with an extra rounding step at the end (right after the PBS). This rounding procedure aims to remove the LSB of the ciphertexts that are different from an implementation to the other. This rounding increases the noise in the plaintext and it adds a new parameter to optimize: the location of the rounding. The higher in the MSB we round the more noise we add, but also the more error we remove. Note that this rounding will either keep the same amount of error (when the output error is maximal, i.e., the ciphertexts are completely different) or cancel it entirely depending on the parameter for the rounding and the input error.

We can add to the optimization framework a new constraint related to the maximum error of the FFT we want to consider. We can do that easily with the additional feasible set $\mathcal{S}_{\mathsf{other}}(\mathcal{G})$. The condition could be represented as: $\mathcal{S}_{\mathsf{other}}(\mathcal{G}) = \{x \in \mathcal{C} | \forall i, \mathbb{E}_{\mathcal{O}_i}(x) = 0\} \subset \mathcal{C}$, with $\mathbb{E}_{\mathcal{O}_i}(x)$ the error of the output ciphertext coefficients of $\mathcal{O}_i$ after the rounding. This approach relies on the fact that we have a model for the error in the output of the PBS in terms of ciphertext coefficients.

Some cryptographic observations can help reducing the size of the parameter space. In particular, we must have: $\frac{q}{2\beta_{\mathsf{PBS}}^{\ell_{\mathsf{PBS}}}} > \mathsf{error}_{FFT}(x)$.

This optimization enables to set a limit in terms of error in the ciphertext coefficients that an implementation of the FFT introduces. Then, we can optimize for a given FFT error model, some noise model, and some cost model targeting a common architecture for instance. The result of the optimization can be used to compute the same circuit on the same ciphertexts with the same public keys, but with different implementation of the same FHE algorithms and we will end up with the exact same ciphertext in output.

This feature enables many miners in a blockchain for instance, to compute the same circuit on the same inputs and have a consensus without a need to decrypt anything. It guarantees that the result came out of the desired FHE DAG and not another designed by an attacker.

Optimization for Several Public Keys In previous applications, we assumed that we have only one public material per FHE operator for the whole FHE DAG.

Restricting the number of public keys helps to have a small quantity of public material. It also has an impact on the complexity of the optimization problem because as a result, parameters are shared across the entire FHE DAG. The down side is that we cannot speed up parts of the FHE DAG that have a bigger noise bound or less leveled operations (smaller 2-norm) with smaller and faster parameters.

In this section we describe a simple optimization problem: one LWE secret key $\vec{s}$ and one GLWE secret key $\vec{S'}$ (for the PBS) that can be viewed as a bigger LWE secret key $\vec{s'}$, along with X different key switching keys going from $\vec{s'}$ to $\vec{s}$. These key switching keys can use a different base β and/or a different number of levels ℓ. We consider a graph $\mathfrak{G}$ of Y APs of type #1.

There are many ways to solve this problem. A naive solution is to consider different parameters for each KS and to let every KS to have its own (β, ℓ) and to add the constraint that they can have at most X values. This approach increases exponentially the search space of the optimization problem, so we will not consider it.

A second solution, which is straightforward though not the most efficient one, is to introduce a new variable δ for each KS. This value δ stores the associated KSK identifier. This is a new parameter to optimize for each KS: $\forall i \in [1, Y]\, \delta_i \in [1, X]$. So our additional search space, defined by the problem of finding which KSK is used by which KS, is of size X^Y.

We designed a third solution to solve the problem. Starting now, we will sort our KSK $(\mathsf{KSK}_0, \mathsf{KSK}_1, \cdots)$ such that a KS with KSK_i adds less noise than using KSK_{i+1} for all $0 \leq i$. Let's consider the following toy example: $t \in \mathbb{N}$ is a noise bound and $\mathfrak{G} = \{A_0(\nu_0, t), A_1(\nu_1, t), A_2(\nu_2, t)\}$ is an FHE DAG such that $\nu_0 < \nu_1 < \nu_2$. This graph has the same noise bound t for each AP and they are all of type #1.

Let us illustrate this method with three APs and two possible keyswitching keys and let us assume that $\vec{\delta} = (\delta_0, \delta_1, \delta_2) = (0, 1, 0)$ is the optimal solution, we will show that it cannot be so. We can use Theorem 3 to infer that $\mathcal{S}_{\mathbb{N}}(A(\nu_2, t)) \subseteq \mathcal{S}_{\mathbb{N}}(A(\nu_1, t)) \subseteq \mathcal{S}_{\mathbb{N}}(A(\nu_0, t))$, and Theorem 4 to infer that a KS with KSK_1 is faster

than a KS with KSK_0. It is then straightforward to see that if $(0, 1, 0)$ is a solution, then $(1, 1, 0)$ is also a solution but a faster one. This example can be extended to an arbitrary number of AP sharing the same noise bound.

To speed up the optimization problem, as pre-computation, we sort the AP according to their 2-norm ν_i. We will also define each indexes γ_i of $\vec{\delta}$ where we stop to have one of the possible KSK, i.e. for all $0 \leq i \leq X-1$ we have a $\gamma_i \in \{0, \cdots, Y\}$ such that $\delta_{\gamma_i} < i$ and $\delta_{\gamma_i - 1} \geq i$. It enables to have less variables to optimize: instead of Y variables, one for each keyswitch, we only have X variables the number of authorized KSK.

We can consider an FHE DAG with different noise bounds, and apply what we just described for each of the different noise bounds. The same approach works to enable several BSK in the optimization. Indeed, BSK can also be sorted by amount of noise they offer in their output ciphertexts.

Faster Computation from Split DP In this application we introduce another strategy to end up with faster computations. The idea is to enable to break DP operators into several smaller DPs by inserting some PBS between them.

We consider a graph $\mathcal{G} = \{A(\nu_i, t)\}_{0 \leq i < \alpha}$ composed of AP of type #1 which involves a DP operator. We introduce a new AP type and translate every AP of type #1 into this new AP type $A^* \in \mathcal{A}^{(1^*)}$ which has the exact same sub-graph than AP type #1 except that the DP is split into several sub-DP connected with PBS as explained in Theorem 6. This AP has a new parameter d_i describing the splitting of the DP for the i-th AP, i.e., how many sub-DP we will have. Note that a graph $\mathcal{G}^*$ with fixed values d_i can be viewed as a graph $\widetilde{\mathcal{G}^*}$ of AP type #1.

Formally, the problem can be written this way: $\arg\min_{\vec{d} \in \mathcal{D}} \mathsf{Cost}\left(\mathcal{G}^*_{\vec{d}}, x^*\right)$ where $\mathcal{G}^*_{\vec{d}} = \{A^*(\nu_i, t, d_i)\}_{0 \leq i < \alpha}$, and x^* is the optimal solution obtained as in the Section 5.1 over the $\vec{d}$-split graph $\mathcal{G}^*$, i.e. $x^* = \arg\min_{x \in \mathfrak{C}} \mathsf{Cost}\left(\widetilde{\mathcal{G}^*}, x\right)$ and $x \in \mathcal{S}_{\mathbb{N}}\left(\widetilde{\mathcal{G}^*}\right)$ where $\mathfrak{C}$ is the search space of $\widetilde{\mathcal{G}^*}$.

Several ways can be imagined to split a DP. One could be to group public weights of the DP into d_i sets such that their 2-norm is approximately the same. This will yield the best result if we keep neglecting the cost of the DP. Inserting a PBS adds extra operations to perform, but it will also reduce the 2-norm of the initial DP. This may lead to noisier parameters, but faster and still as correct. To sum up, this method allows to optimally insert PBS during leveled operations, here a dot product.

We describe a few other applications in Supplementary Material because of lack of space. In particular we explain how to optimize with our framework when we mix different AP types (Supplementary Material E.2). We also investigate the impact that has the FFT in an FHE operator, with the example of the PBS in Supplementary Material E.3.

6 Conclusion & Future Work

Finding parameters that are correct, secure and efficient is a hard problem that hinders large scale adoption of FHE. In this paper, we proposed the first optimization framework that allows to efficiently select the best FHE parameters for TFHE-like schemes given a *plain graph*, a cost and a noise models. It allowed us to compare several of the bootstrapping algorithms described in the literature and a new WoP-PBS (PBS without padding) in diverse scenarios. As a result, we know for each of them, the contexts where they are suited the most. This knowledge will be useful to accelerate an optimization process that has many choices in terms of bootstrapping algorithms.

We also proposed new types of ciphertexts, combining several LWE encryptions to encode large precision messages by using a radix-based and/or as a CRT-based decomposition. We used our new optimization framework to provide optimal parameters, for those large homomorphic integers as well as for other practical applications.

Future Work An interesting future work would be to extend the optimization to more than the cryptographic parameters. For instance, many new parameters comes with the large homomorphic integer representation we introduced in this paper, such as the moduli or the number of blocks, and they could also be optimized. A second example is to optimize the topology of the graph of FHE operators by offering to the optimization many different options. Another future work would be to use our new optimization framework to find optimal parameters for more FHE schemes, other than the TFHE-like ones.

References

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 2015.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *ITCS*, 2012.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. *IACR Cryptology ePrint Archive*, 2012.

[BST20] Florian Bourse, Olivier Sanders, and Jacques Traoré. Improved secure integer comparison via homomorphic encryption. In *CT-RSA*. Springer, 2020.

[CCR19] Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring ORAM: efficient constant bandwidth oblivious RAM from (leveled) TFHE. In *CCS 2019*. ACM, 2019.

[CDS15] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. Armadillo: a compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, 2015.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *J. Cryptol.*, 2020.

[CIM19] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. New techniques for multi-value input homomorphic evaluation and applications. In *CT-RSA*. Springer, 2019.

[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 *CSCML 2021*. Springer, 2021.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT 2017*, 2017.

[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.

[CMG+18] Eduardo Chielle, Oleg Mazonka, Homer Gamil, Nektarios Georgios Tsoutsos, and Michail Maniatakos. E3: A framework for compiling c++ programs with encrypted operands. Cryptology ePrint Archive, Paper 2018/1013, 2018.

[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 *EUROCRYPT 2015*, 2015.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012.

[GBA21] Antonio Guimarães, Edson Borin, and Diego F. Aranha. Revisiting the functional bootstrap in TFHE. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *CRYPTO 2013*. Springer, 2013.

[KO22] Jakub Klemsa and Melek Onen. Parallel operations over tfhe-encrypted multi-digit integers. Cryptology ePrint Archive, Report 2022/067, 2022.

[KS21] Kamil Kluczniak and Leonard Schild. FDFB: full domain functional bootstrapping towards practical fully homomorphic encryption. *CoRR*, 2021.

[LMP21] Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. Large-precision homomorphic sign evaluation using fhew/tfhe bootstrapping. Cryptology ePrint Archive, Report 2021/1337, 2021. https://ia.cr/2021/1337.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT 2010*. Springer, 2010.

[Pad11] David Padua. *FFTW*, pages 671–671. Springer US, Boston, MA, 2011.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC 2005*. ACM, 2005.

[SSTX09] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In *ASIACRYPT 2009*. Springer, 2009.

[VJH21] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. Sok: Fully homomorphic encryption compilers. *CoRR*, 2021.

Supplementary Material

A Details on Advanced Modular Arithmetic from Single LWE Ciphertexts

In this section we provide more details on the advanced modular arithmetic operations from Section 4.2.

A.1 Arithmetic Operators

Thanks to the carry subspace, we can compute leveled operations such as homomorphic additions or multiplications with a known constant. Let's consider two LWE ciphertexts ct_1 and ct_2 encrypting respectively m_1 and m_2 with respective degrees of fullness $\deg_1$ and $\deg_2$. From the degree of fullness one can infer respective worst case messages μ_1 and μ_2 (Definition 14). The following operations are allowed as long as both ciphertexts share the same base β, carry-message modulus p and ciphertext modulus q. Our plaintext format does not allow any native modular reduction modulo β, i.e. the carry subspace will contain the quantity overlapping β and we cannot have a degree of fullness greater than 1 at any time for correctness reason.

Addition To compute the addition $m_1 + m_2$ modulo β one can use the traditional LWE addition. With this approach, the necessary condition to guarantee correctness is $\deg_1 + \deg_2 \leq 1$ and the output ciphertext will have a degree equal to $\deg_1 + \deg_2$.

Multiplication To compute a multiplication between ct_1 (as defined above) and an integer constant $0 \leq c$ one can use the trivial multiplication between an LWE ciphertext and a positive integer. With this approach, the necessary condition to guarantee correctness is $c \cdot \deg_1 \leq 1$ and the output ciphertext will have a degree equal to $c \cdot \deg_1$.

Opposite To compute the opposite of m_1 modulo β we can use the trivial algorithm where we compute the opposite of every elements of the LWE ciphertext ct_1. However, without a correction, this will lead to an encoding of a message that is no more between 0 and $p-1$. This is why one must add the correction term $\beta \cdot \left\lceil \frac{\mu_1}{\beta} \right\rceil$ after computing the opposite of the each coefficients of the ciphertext. With this approach, the necessary condition to guarantee correctness is $\beta \cdot \left\lceil \frac{\mu_1}{\beta} \right\rceil \leq p-1$ and the output degree of fullness is $\beta \cdot \left\lceil \frac{\mu_1}{\beta} \right\rceil \cdot \frac{1}{p-1}$.

Subtraction To compute the subtraction $m_1 - m_2$ modulo β one starts by homomorphically compute the opposite of ct_2 and then compute the homomorphic addition with ct_1. Both the condition and the output degree of fullness can be infered from the descriptions of the previous operations.

LUT Evaluation The PBS is an operation that allows to evaluate a uni-variate function on the input at the same time as it reduces the noise. It is then easy to compute $l(m_1)$ homomorphically from ct_1 with l a LUT. The requirement is that $\deg_1 \leq 1$, and the output degree is $\frac{\mu_l}{p-1}$ where μ_l is the biggest possible output of the LUT.

A.1.1 PBS with p Not a Power of Two

No details, nor analysis have been provided yet in the literature about computing programmable bootstrapping when the plaintext space is not a power of two. It is required to have a bit of padding when one wants to compute a traditional PBS evaluating a non-negacyclic function, which forces the plaintext space to be even. However the algorithm works the same with an odd p, the only difference lies in the way the r-redundant LUT is build. This also brings a slight modification in the evaluation of the error probability when computing such PBS.

Recall that such LUT are encoded in the polynomial plaintext $\tilde{L} = \mathsf{Encode}(L, p', q)$ of a GLWE ciphertext and L contain redundancy. We call *mega-cases* each block of successive redundant values. In a generic manner, if the LUT we want to compute is defined as $L : \mathbb{Z}_p \to \mathbb{Z}_{p'}$, $x \mapsto y_x$, we define the polynomial L as:

$$L = X^{-\left\lfloor \frac{N}{2 \cdot p} \right\rceil} \cdot \left(\sum_{i=0}^{N-1} y_{\left\lfloor \frac{i \cdot p}{N} \right\rceil} \cdot X^i \right)$$

Proof 6 (Sketch) *With such a LUT, and p not a power of 2, we end up 2 possible sizes for the mega-cases of the r-redundant LUT: either $\left\lfloor \frac{N}{p} \right\rfloor$ or $\left\lceil \frac{N}{p} \right\rceil$. For the correctness study, we will take the worst case scenario, i.e. considering $\left\lfloor \frac{N}{p} \right\rfloor$. The encoding function (Definition 2) enables to have messages centered in the mega-cases when it comes to PBS, it means that the probability of going into the wrong mega-case during a PBS in the worst case scenario is when the error e_{MS} is bigger in absolute value than $\left\lfloor \frac{N}{p} \right\rfloor$ where e_{MS} is the error in the PBS after the modulus switch and before the blind rotation. It is easy to estimate e_{MS} as a variance, thanks to noise formulae. Since it is close to a Gaussian distribution, we can use a confidence interval to infer the probability to get into the wrong mega-case.* □

A.2 Multiplications

To compute the addition $m_1 \cdot m_2$ we can use a combination of leveled operations and PBS. Here, we propose three types of multiplication:

(i) the multiplication of the two inputs without any modular reduction returning $m_1 \cdot m_2$ (requiring that $\mu_1 \cdot \mu_2 < p$). The output degree of fullness is $\deg_1 \cdot \deg_2 \cdot (p-1)$;

(ii) the multiplication in the LSB, returning an LWE encryption of $m_1 \cdot m_2 \mod \beta$. The output degree of fullness is the minimum between $\frac{\beta-1}{p-1}$ and $\deg_1 \cdot \deg_2 \cdot (p-1)$;

(iii) the multiplication in the MSB, returning an LWE encryption of $\left\lfloor \frac{m_1 \cdot m_2}{\beta} \right\rceil$. The output degree of fullness is $\left\lfloor \frac{\mu_1}{\beta} \right\rceil \cdot \left\lfloor \frac{\mu_2}{\beta} \right\rceil$;

The first method can be used to compute a multiplication of type (ii), and is known in the TFHE literature (see as instance [CJL+20]) and consists in computing the multiplication by observing that $x \cdot y = \frac{(x+y)^2}{4} - \frac{(x-y)^2}{4}$ modulo β. Then, we compute $m_1 + m_2$ and $m_1 - m_2$ in a leveled fashion, and we use two KS-PBS with the LUT computing the uni-variate function $\left\lfloor \frac{x^2}{4} \right\rceil \mod \beta$ to compute $\frac{(m_1+m_2)^2}{4} \mod \beta$ and $\frac{(m_1-m_2)^2}{4} \mod \beta$. We finally subtract the two results and perform another KS-PBS with LUT computing $x \mod \beta$ to get the right result.

The second method we propose is using the **ChainedPBS**. It can be used to compute multiplications of type (i), type (ii) and type (iii), and requires the use of the technique presented in one of the previous paragraphs and illustrated in Figure 13.

We evaluate the multiplication of type (ii), as a bi-variate function, by shifting one of the two ciphertexts, adding to the other one and by performing a KS-PBS with LUT computing the function $(x \mod (\mu_1 + 1)) \cdot \left(\left\lfloor \frac{x}{\mu_1+1} \right\rceil \mod \beta \right) \mod \beta$. We evaluate the multiplication of type (iii) in the same manner. The only difference is that the KS-PBS evaluates a LUT computing the function $\left\lfloor \frac{(x \mod (\mu_1+1)) \cdot \left(\left\lfloor \frac{x}{\mu_1+1} \right\rceil \mod \beta \right)}{\beta} \right\rceil$. We evaluate the multiplication of type (i) in the same manner. The only difference is that the KS-PBS evaluates a LUT computing the function $(x \mod (\mu_1 + 1)) \cdot \left\lfloor \frac{x}{\mu_1+1} \right\rceil$.

The third method can be use to compute the multiplication of type (i). It consists on using a BFV GLWE multiplication as introduced in [CLOT21] for the TFHE context.

A.3 Carry & Message Extractions

As we observed in previous sections, after performing homomorphic operations the degree of fullness increases and the carry subspace might need to be emptied. To do so, we propose two operations: the *carry extract* operation, that allows to extract the carry $\left\lfloor \frac{m_1}{\beta} \right\rceil$ of m_1 overlapping β into a new LWE ciphertext (one or more), and *message extract* operation, which allows to extract $m_1 \mod \beta$ into a new LWE ciphertext.

Both operations use a PBS (along with key switching in order to come back to the original secret key, if necessary), taking as input the same LWE ciphertext and the same public material (i.e., the bootstrapping and key switching keys), but different LUT. The carry extract uses $P_{\mathsf{carry\text{-}ext}}$: a r-redundant LUT for $x \to \left\lfloor \frac{x}{\beta} \right\rfloor$. The output degree of fullness is $\left\lfloor \frac{\mu_1}{\beta} \right\rfloor$. The message extract uses $P_{\mathsf{msg\text{-}ext}}$: a r-redundant LUT for $x \to x \mod \beta$. The output degree of fullness is the minimum between $\frac{\beta-1}{p-1}$ and $\deg_1$. If the carry subspace is larger than the message subspace, more than one PBS might be required to empty it all along with slightly different LUTs.

B Example on Radix-Based Integers

In this section we give more detail about the homomorphic large integers introduced in this paper.

Multiplication Let's use a toy example to describe how a multiplication between two encrypted radix-based modular integers. We will use $\kappa = 2$, $\beta_0 = 3$, $\beta_1 = 5$ and $\Omega = 15$. We will multiply $\mathsf{msg} = \mathsf{msg}_1 \cdot \mathsf{msg}_2$ modulo 15 with $\mathsf{msg}_1 = 10 = 3 \cdot 3 + 1$, $\mathsf{msg}_2 = 5 = 1 \cdot 3 + 2$ and $\mathsf{msg} = m_1 \cdot 3 + m_0$. What we do in clear is $m_0 = 1 \cdot 2 = 2$ and $m_1 = 3 \cdot 2 + 3 \cdot 1 \cdot 3 + 1 \cdot 1 = 16 = 1$ because it lives modulo $\beta_1 = 5$.

We now need to compute this homomorphically between two radix-based modular integers. We will set $p = 32$. We have four (two for each integers) ciphertexts encrypting modular integers: $\mathsf{ct}_1^{(1)}$ encrypting 3 under (β_1, p) with degree $4/31$, $\mathsf{ct}_0^{(1)}$ encrypting 1 under (β_0, p) with degree $2/31$, $\mathsf{ct}_1^{(2)}$ encrypting 1 under (β_1, p) with degree $4/31$, $\mathsf{ct}_0^{(2)}$ encrypting 2 under (β_0, p) with degree $2/31$.

We want to produce a radix-based modular integer that will be composed of two ciphertexts encrypting modular integers: $\mathsf{ct}_1^{(\mathsf{out})}$ under (β_1, p) and $\mathsf{ct}_0^{(\mathsf{out})}$ under (β_0, p). The computation will be the following: $\mathsf{ct}_0^{(\mathsf{out})} \leftarrow \mathsf{Mul}\left(\mathsf{ct}_0^{(1)}, \mathsf{ct}_0^{(2)}\right)$ and:

$$\mathsf{ct}_1^{(\mathsf{out})} \leftarrow \mathsf{Mul}\left(\mathsf{ct}_0^{(1)}, \mathsf{ct}_1^{(2)}\right) + \mathsf{Mul}\left(\mathsf{ct}_1^{(1)}, \mathsf{ct}_0^{(2)}\right) + \mathsf{LUT\text{-}Eval}\left(x \mapsto \beta_0 \cdot x \mod \beta_1, \mathsf{Mul}\left(\mathsf{ct}_1^{(1)}, \mathsf{ct}_1^{(2)}\right)\right)$$

Note that Mul refers to the multiplication of type (i), and that $\mathsf{LUT\text{-}Eval}$ refers to the LUT evaluation in the same section and its first input is a description of the LUT to evaluate. The degree of fullness of $\mathsf{ct}_0^{(\mathsf{out})}$ is $4/32$, and the degree of $\mathsf{ct}_1^{(\mathsf{out})}$ is $8/32 + 8/32 + 4/32 = 20/32$. They will encrypt respectively $1 \cdot 2 = 2$ and $1 \cdot 1 + 3 \cdot 2 + (3 \cdot 1 \cdot 3 \mod 5) = 11$ which when decoding will lead to the expected value.

We could definitely have computed the same functionality, i.e. $\mathsf{msg}_1 \cdot \mathsf{msg}_2$, from the other two types of multiplication, and it would have only changed a few details in the algorithm. Also note that we computed $\mathsf{LUT\text{-}Eval}$ instead of using a simple multiplication with a constant so the output degree does not become too big. Here again there are many different combination of parameters and algorithms that would have produced the desired result. We provide details on this technique in Algorithm 3 in Supplementary Material D.

Opposite For the negation, we can use msg_1 defined above. We start by computing the opposite of the MSB block, i.e. $\mathsf{ct}_1^{(1)}$, so we end up with the message $5-3=2$. Then we need to compute the **opposite** of the LSB block, i.e. $\mathsf{ct}_1^{(1)}$, so we end up with the message $3-1=2$, and finally compensate this LSB opposite by subtracting 1 to the MSB block. It means that we will end up with $2+(5-1)=6$ encrypted in the MSB block. After decoding, it will output the expected value.

LUT Evaluation There are many examples of univariate functions (such as the inverse). These functions could be computed with the techniques we just described and then, by using a similar method than the one we used to compute the multiplication, it is also possible to compute an homomorphic division of the form $\left\lfloor \frac{m_1}{m_2} \right\rceil$.

C Details on the Hybrid Approach

As we explained above, the CRT-only approach has some limitations. To overcome them, we create a new homomorphic hybrid representation that mixes the CRT-based approach (Section 4.4) with the radix-based approach (Section 4.3), in order to take advantage of the best of both worlds. The idea is to use the CRT approach as the top layer in the structure, and to represent the CRT residues by using radix-based modular integers when needed: with this approach we do not have any more restrictions on Ω.

Let $(\Omega_0, \cdots, \Omega_{\kappa-1})$ be integers co-primes to each other, i.e., (Ω_i, Ω_j) co-primes for all $i \neq j$, and let $\Omega = \prod_{i=0}^{\kappa-1} \Omega_i$.

Encode To encode a message $\mathsf{msg} \in \mathbb{Z}_\Omega$, as we do in the CRT-only approach, we split into a list of $\{\mathsf{msg}_i\}_{i=0}^{\kappa-1}$ such that $\mathsf{msg}_i = \mathsf{msg} \mod \Omega_i$ for all $0 \leq i < \kappa$. At this point, for each message msg_i for $i \in [\![0, \kappa-1]\!]$, we apply the same encoding used for radix-based modular integers, as in Section 4.4 (Encode from Definition 2). It means that any CRT residue Ω_i have its own list of radix bases: $(\beta_{i,\kappa_i-1}, \cdots, \beta_{i,0})$ and more generally its parameters $\{(\beta_{i,j}, p_{i,j})\}_{0 \leq j < \kappa_i} \in \mathbb{N}^{2\kappa_i}$. Figure 11 in Supplementary Material H gives a visual representation.

Decode The decoding is done in two steps: first, we decode each independent radix-based modular integer to obtain the independent residues modulo $\Omega_0, \ldots, \Omega_{\kappa-1}$, and then we invert the CRT to retrieve the message modulo Ω.

The hybrid approach can be seen as a generalization of both the CRT-only approach (if $\kappa_i = 1$ for all $0 \leq i < \kappa$) and the pure radix-based modular integer approach (if $\kappa = 1$). It also covers the mixed cases where some of the κ_i are equal to 1 and the others are greater.

We say that two hybrid ciphertexts are compatible to perform homomorphic operations if they are defined under the same CRT residues and for each residue, the same parameters. To perform any homomorphic operation, we have to perform

the computation on each radix component independently, as shown for the CRT-only approach. The only difference here is that, instead of manipulating a single LWE ciphertext (as in Section 4.2), we manipulate radix-based modular integers (as in Section 4.4).

D Detail about Algorithms

In this section we provide algorithms that are used in the main body of this paper.

Let $\vec{\beta} = (\beta_0, \ldots, \beta_{\kappa-1})$ and $\vec{p} = (p_0, \ldots, p_{\kappa-1})$. We need to define recursively the quantities $q_{i,\vec{\beta}}$, $r_{i,\vec{\beta}}$ and $\gamma_{\vec{\beta}}$, for $i \in \mathbb{Z}_{\geq -1}$, which are useful in these algorithms, as:

- $q_{i,\vec{\beta}}(x) = \begin{cases} x, \text{ if } i = -1 \\ \left\lfloor \frac{q_{i-1,\vec{\beta}}(x)}{\beta_i} \right\rfloor, \text{ if } i \geq 0 \end{cases}$
- $r_{i,\vec{\beta}}(x) = q_{i-1,\vec{\beta}}(x) - q_{i,\vec{\beta}}(x) \cdot \beta_i, \ i \geq 0$
- $\gamma_{\vec{\beta}}(x) = \begin{cases} \min(i \in \Omega), \ \Omega = \{i < |\vec{\beta}|, \ q_{i,\vec{\beta}}(x) = 0\} \\ |\vec{\beta}| \text{ if } \Omega = \varnothing. \end{cases}$

The first algorithm we need is the Decomposition algorithm: we give details in Algorithm 2.

The second tool we need is a padding algorithm, which allows to change the size of a radix-based large integer encryption (Section 4.3) from κ to κ_{out}. To do so, we complete the ciphertext with trivial encryptions of zero. This is useful when we use the Decomp Algorithm 2 since the output ciphertext might not be of length κ.

The signature of the algorithm is:

$$\boxed{\left(c'_0, \ldots, c'_{\kappa_{\mathsf{out}}-1}\right) \leftarrow \mathsf{Pad}\left(\left(c_0, \ldots, c_{\kappa-1}\right), \alpha, \vec{p}_{\mathsf{out}}, \vec{\beta}_{\mathsf{out}}\right)}$$

We give details on the multiplication operation for radix-based modular integers in Algorithm 3.

Algorithm 3: $\left(\mathsf{ct}_0^{(\mathsf{out})}, \ldots, \mathsf{ct}_{\kappa-1}^{(\mathsf{out})}\right) \leftarrow \mathsf{Mult}\left(\left(\mathsf{ct}_0^{(1)}, \ldots, \mathsf{ct}_{\kappa_1-1}^{(1)}\right), \left(\mathsf{ct}_0^{(2)}, \ldots, \mathsf{ct}_{\kappa_2-1}^{(2)}\right), \mathsf{PUB}\right)$

Context:
$$\begin{cases} (q, p_{j,i}, \beta_{j,i}, \deg_{j,i}): \text{ paremathers of } \mathsf{ct}_i^{(j)} \\ \mu_{j,i} := \deg_{j,i} \cdot (p_{j,i} - 1) + 1,\ j \in [\![1,2]\!],\ i \in [\![0, \kappa_j - 1]\!] \\ \vec{\beta}_j := (\beta_{j,0}, \ldots, \beta_{j,\kappa_j-1}),\ \vec{p}_j := (p_{j,0}, \ldots, p_{j,\kappa_j-1}) \\ \vec{\beta}_{j,i} := (\beta_{j,i}, \ldots, \beta_{j,\kappa_j-1}),\ \vec{p}_{j,i} := (p_{j,i}, \ldots, p_{j,\kappa_j-1}) \\ \gamma_{h,k} := \gamma_{\overrightarrow{\beta}_{2,k}}((\mu_{1,h} - 1) \cdot (\mu_{2,k} - 1)) \\ \{P_{i,r_k,\overrightarrow{\beta_{2,j}}}\}_{0 \le i \le \kappa_1 - 1,\ 0 \le j \le \kappa_2 - 1,\ 0 \le k \le \gamma_{i,j}} : \text{a LUT for} \\ x \to r_{k,\overrightarrow{\beta}_{2,j}}\left((x \mod \mu_{2,i}) \cdot \left\lfloor \frac{x}{\mu_{2,i}} \right\rfloor\right) \cdot \frac{q}{2 \cdot p_{2,k+j}} \end{cases}$$

Input:
$$\begin{cases} \left(\mathsf{ct}_0^{(i)}, \ldots, \mathsf{ct}_{\kappa_i-1}^{(i)}\right): \text{ encrypting } \mathsf{msg}_i \text{ under } \vec{s},\ i \in [\![1,2]\!] \\ \mathsf{PUB}\text{: public material for KS-PBS} \end{cases}$$

Output: $\left(\mathsf{ct}_0^{(\mathsf{out})}, \ldots, \mathsf{ct}_{\kappa-1}^{(\mathsf{out})}\right)$ encrypting $\mathsf{msg}_1 \cdot \mathsf{msg}_2$ under $\vec{s}$

1 **for** $i \in [\![0; \kappa_1 - 1]\!]$ **do**
  `/* Put the block to the right basis */`
2 $\mathsf{ct}_{tmp} \leftarrow \mathsf{LweBasisChange}(\mathsf{ct}^{(1)}i, p_{2,0}, \beta_{2,0}, \mathsf{PUB})$
  `/* Compute the multiplication-decomposition */`
3 $\left(\mathsf{ct}_0^{(tmp)}, \ldots, \mathsf{ct}_{\kappa-1}^{(tmp)}\right) \leftarrow \mathsf{OneBlockMul}\left(\mathsf{ct}_{tmp}, \left(\mathsf{ct}_0^{(2)}, \ldots, \mathsf{ct}_{\kappa_2-1}^{(2)}\right), \mathsf{PUB}\right)$
  `/* Add the results of the multiplications together */`
4 $\left(\mathsf{ct}_0^{(\mathsf{out})}, \ldots, \mathsf{ct}_{\kappa-1}^{(\mathsf{out})}\right) \leftarrow \mathsf{Add}(\left(\mathsf{ct}_0^{(\mathsf{out})}, \ldots, \mathsf{ct}_{\kappa-1}^{(\mathsf{out})}\right), \left(\mathsf{ct}_0^{(tmp)}, \ldots, \mathsf{ct}_{\kappa-1}^{(tmp)}\right))$
5 **end**
6 **return** $\left(\mathsf{ct}_0^{(\mathsf{out})}, \ldots, \mathsf{ct}_{\kappa-1}^{(\mathsf{out})}\right)$

We report details on the two methods we propose for the modular reduction in Algorithm 4 and Algorithm 5.

Observe that the κ KS-PBS in Line 2 of Algorithm 4 could be replaced by optimized procedures evaluating several different LUT on the same input ciphertext. A few constructions have been proposed in the literature, such as the PBS many-LUT [CLOT21] or the multi-value bootstrapping [CIM19].

D.1 Tree PBS approach on Radix-Based Modular Integers

In this section, we give more details on how to apply the TreePBS technique by [GBA21] to our new radix-based modular integers.

In [GBA21] the plaintext integers are all encrypted under the same basis β: we offer here the possibility to evaluate a large look-up table with integers set in different basis $(\beta_0, \ldots, \beta_{\kappa-1})$.

Let $\Omega = \prod_{i=0}^{\kappa-1} \beta_i$, and let $L = [l_0, l_1, \cdots, l_{\Omega-1}]$ be a LUT with Ω elements. We want to evaluate this LUT on a radix-based modular integer encrypting a message $\mathsf{msg} = m_0 + \sum_{i=1}^{\kappa-1} m_i \prod_{j=0}^{i-1} \beta_j$.

Then, to evaluate the new multi-radix tree-PBS we performs the following steps:

1. We note as $B = \{\beta_i | i \in [\![0, \kappa - 1]\!]\}$ and as $\vartheta(\beta_i)$ the component m_i of msg associated to β_i.

2. We define $\beta_{\max} = \max(\beta \in B)$.

3. We split the LUT L into $\nu = \frac{\prod_{\beta_i \in B} \beta_i}{\beta_{\max}}$ smaller LUTs $(L_0, \ldots, L_{\nu-1})$ that each contain $\beta_{\max}$ different elements of L.
4. We compute a PBS on each of the ν LUTs using the ciphertext encrypting $\vartheta(\beta_{\max})$ as a selector.
5. We build a new large look-up table L by packing, with a key switching, the results of the ν iterations of the PBS in previous step.
6. We remove $\beta_{\max}$ from B: $B = B - \beta_{max}$.
7. We repeat the steps from 2 to 6 until B is empty.

The generalized multi-radix tree-PBS takes as input a radix-based modular integer ciphertext, a large look-up table L and the public material required for the PBS and key switchings and returns a LWE ciphertext. The signature is: $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{Tree\text{-}PBS}((\mathsf{ct}_0, \ldots, \mathsf{ct}_{\kappa-1}), \mathsf{PUB}, L)$.

For the CRT-only and hybrid approaches, the multi-radix tree-PBS works in the same way.

E Details on Optimization

In this section we give more detail about the optimization framework.

E.1 AP Splitting

To split a DP in three it translates into the following: $\min_{w_{i,1},\Lambda_1,\Lambda_2} \max(\nu_1, \nu_2, \nu_3)$ such that $w_i = w_{i,1}\Lambda_1 + w_{i,2}\Lambda_2 + w_{i,3}$ for all $j \in \{1, 2\}$, $\mathsf{PGCD}\,(w_{i,j}, \Lambda_j) = 1$ and for all $j \in \{1, 2\}$, $\nu_j = \sqrt{\sum_i w_{j,1}^2}$. This is easy to generalize.

Let $w_i \in [-2^p, 2^p]$ the weights of the DP. If they follows a uniform distribution, we have a trivial way yet very efficient (regarding the noise) to split this DP. We can radix-decompose all the DP weights with the level being equal to $d+1$ and the $\log_2$ of the base β is equal to $\frac{p+1}{d+1}$. Here each Λ_i is equal to a power of β.

E.2 Mixing Different AP Types in an FHE Graph

We consider an FHE graph $\mathfrak{G}$ containing two types of AP: type #1 and type #2. We can apply what we did previously independently on every AP types. At the end of this procedure, we end up with a few AP of type #1 and a few AP of type #2. In practice we have at most as many as we have different noise bounds for both types of AP type.

We can reduce the number of AP needed to compute the noise feasible set $\mathcal{S}_{\mathbb{N}}(\mathfrak{G})$ by comparing AP even when they do not share the same type thanks to Theorem 3.

E.3 Study of the FFT's Impact on Parameters

The FFT makes the PBS computation fast in the TFHE context. However, it adds some error, due to the (double) floating points representation and the very nature of the Fourier domain. We know that it is not negligible when we work with larger precisions but it was never investigated. Our optimization framework enables to understand the impact to use the FFT in an FHE operator. In this application we will focus on the PBS: we want to see what happens to the cost of AP type #1, with and without the additional noise brought by the FFT, and while assuming the same cost in both cases. AP type #6 is needed. It is an imaginary concept since it is exactly AP type #1 but without its additional FFT noise and still as fast.

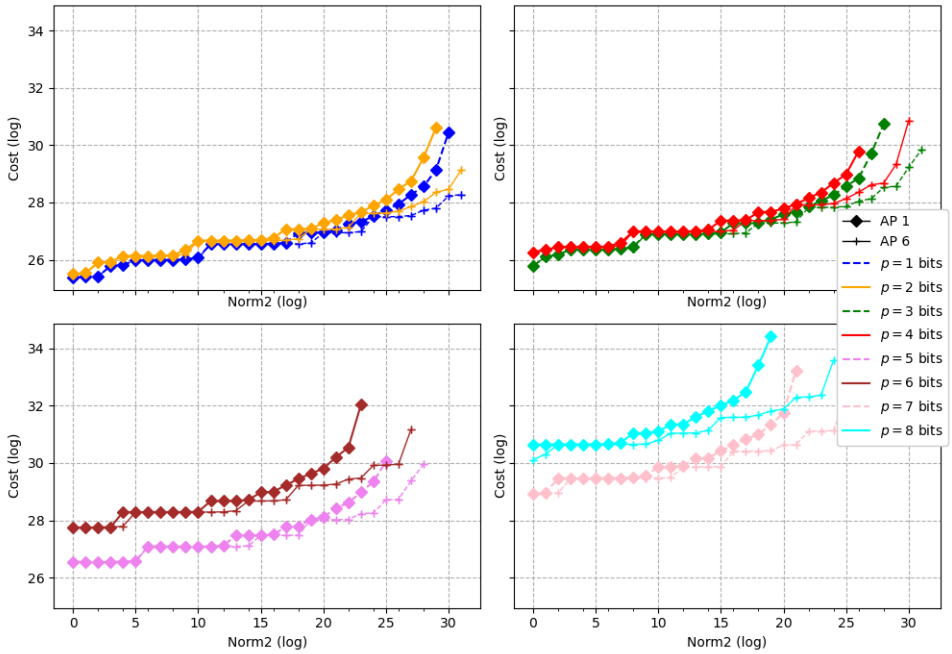

Figure 7: In this figure, we look at the AP of type #1 and type #6. They both follow se same sequence of FHE operators, but the first one has the traditional TFHE PBS (with FFT over f64), and the second one has an imaginary TFHE PBS (with an FFT as costly but with infinite precision). GLWE size is fixed to $k = 1$.

Our results are plotted in Figure 7. We can see that whatever the considered precision is, at some point, the costs of the two AP types diverge when we increase the 2-norm involved in the DP. With high 2-norms, the cost of the AP increases much faster than if our FFT was noiseless. There is a second important observation to make and it is about the existence of parameters. Indeed, for high 2-norms, sooner than with our imaginary FFT, we stop finding possible solutions. To sum up, the higher the precision, the sooner the following phenomenon is observed: not only because of the noise of the FFT there is no solutions for some high 2-norms, the last one possible cost way more.

F Some Benchmarks

In this section, we provide a few benchmarks. The specifications of the machine are: Intel(R) Core(TM) i5-1135G7@2.40GHz, with 16GB of RAM.

G More Optimization Figures

In this section we provide more figures obtained thanks to our optimization framework.

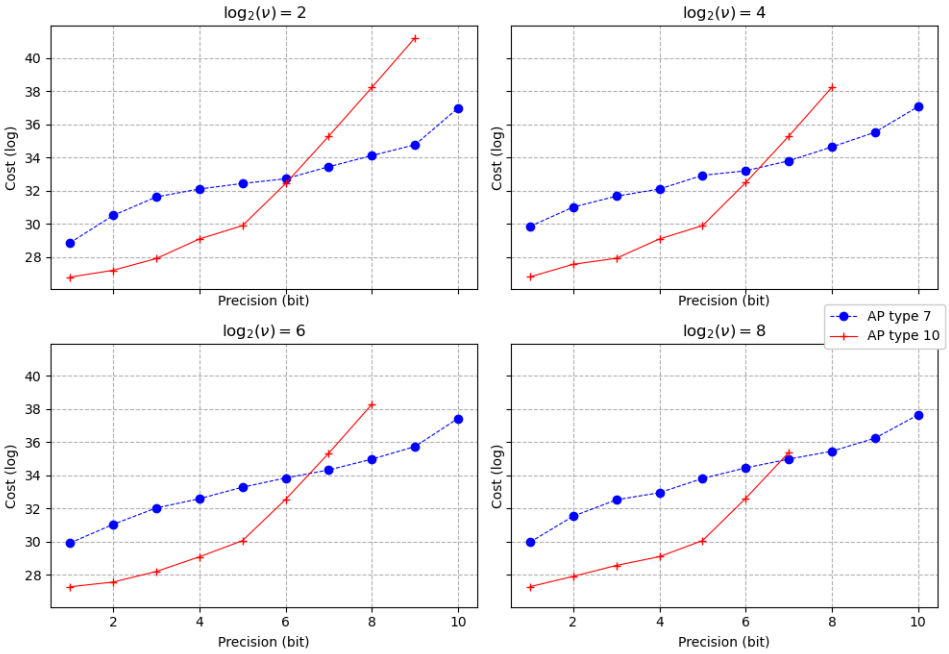

Figure 8: In this figure, we evaluate a LUT over 3 encrypted inputs, with on the one hand our new WoP-PBS (Section 4.1), AP type #7 and on the other hand with the tree-PBS [GBA21], AP type #10.

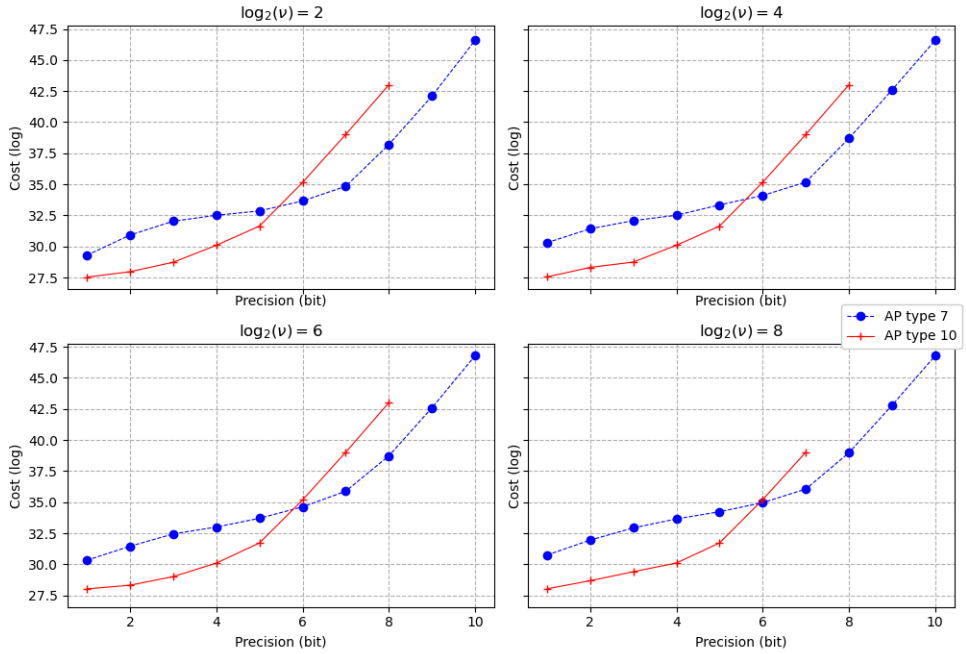

Figure 9: In this figure, we evaluate a LUT over 4 encrypted inputs, with on the one hand our new WoP-PBS (Section 4.1), AP type #7 and on the other hand with the tree-PBS [GBA21], AP type #10.

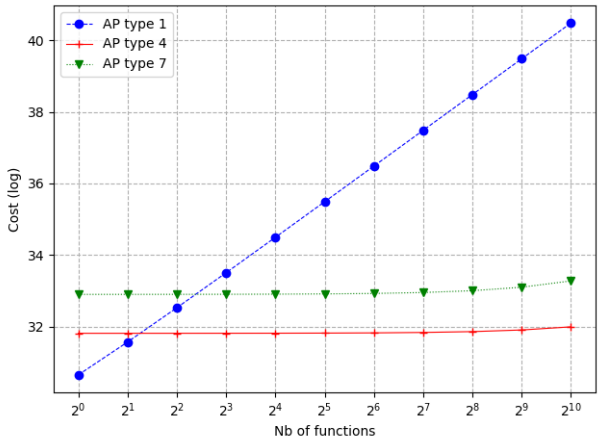

Figure 10: In this figure, we compare AP of type #7, #1, and #4. The first one corresponds to DP-KS followed by our new WoP-PBS (Section 4.1), the second one to DP-KS-PBS, and the last one to DP-KS followed by a multi-value PBS introduced in [CIM19]. Precision is fixed to 8 bits and the 2-norm to 2 bits.

H More Homomorphic Integer Related Figures

In this section, we provide extra figures related to our homomorphic large integers.

$$\mathsf{msg} \mod \Omega \mapsto \begin{cases} \mathsf{msg}_0 = \mathsf{msg} \mod \Omega_0 \mapsto \begin{cases} \{m_{0,j}\}_{j=0}^{\kappa_0-1} \text{ s.t.} \\ \mathsf{msg}_0 = m_{0,0} + \sum_{j=1}^{\kappa_0-1} m_{0,j} \cdot \left(\prod_{k=0}^{j-1} \beta_{0,k}\right) \\ \text{and } \tilde{m}_{0,j} = \mathsf{Encode}\left(m_{0,j}, p_{0,j}, q\right) \\ \forall 0 \leq j < \kappa_0 \end{cases} \\ \vdots \\ \mathsf{msg}_{\kappa-1} = \mathsf{msg} \mod \Omega_{\kappa-1} \mapsto \begin{cases} \{m_{\kappa-1,j}\}_{j=0}^{\kappa_{\kappa-1}-1} \text{ s.t.} \\ \mathsf{msg}_{\kappa-1} = m_{\kappa-1,0} + \sum_{j=1}^{\kappa_{\kappa-1}-1} m_{0,j} \cdot \left(\prod_{k=0}^{j-1} \beta_{\kappa-1,k}\right) \\ \text{and } \tilde{m}_{\kappa-1,j} = \mathsf{Encode}\left(m_{\kappa-1,j}, p_{\kappa-1,j}, q\right) \\ \forall 0 \leq j < \kappa_{\kappa-1} \end{cases} \end{cases}$$

Figure 11: Hybrid approach visualisation combining CRT representation on the top level and radix representation below.

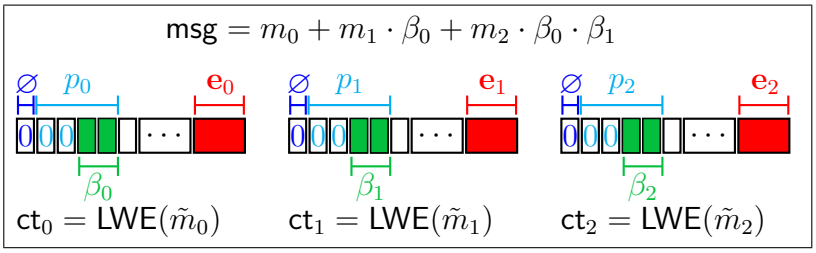

Figure 12: Plaintext representation of a fresh radix-based modular integer of length $\kappa = 3$ working modulo $\Omega = (2^2)^3$. The symbol $\varnothing$ represents the padding bit needed for the PBS. For each block we have $\tilde{m}_i = \mathsf{Encode}\left(m_i, p_i, q\right)$. For all $0 \leq i < \kappa$ we have $\beta_i = 4$, $p_i = 16$, $\kappa = 3$ and $\Omega = 4^3$.

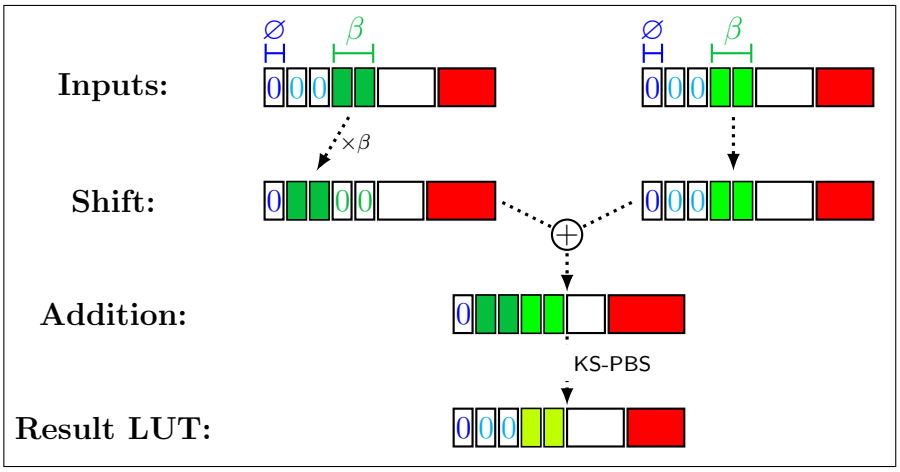

Figure 13: Example of a bi-variate LUT evaluation with shift and PBS.

I Details on Crypto Parameters

In this section we provide parameters obtained with our optimization framework.

I.1 Parameters for Radix-Based Integers

We report in Table 4 the parameters for radix-based integers.

I.2 Parameters for CRT-Based Integers

We used $\Omega_{\mathsf{CRT},16\text{ bits}} = 2^3 \cdot 3^2 \cdot 11 \cdot 13 \cdot 7$ and we have $\log_2(\Omega_{\mathsf{CRT},16\text{ bits}}) = 16.137151260278305$. We used $\Omega_{\mathsf{CRT},32\text{ bits}} = 43 \cdot 47 \cdot 37 \cdot 7^2 \cdot 29 \cdot 41$ and we have $\log_2(\Omega_{\mathsf{CRT},32\text{ bits}}) = 32.02054981586955$.

We use in table 1 the parameters under the crypto ID 9:

- $\log_2(p) = 4$
- $\log_2(\nu) = 8$
- $\sigma^2_{\mathsf{KSK}} = 4.168323308734758e-10$
- $\sigma^2_{\mathsf{BSK}} = 4.905643852600863e-32$
- $k = 2$
- $\log_2(N) = 10$
- $n = 667$
- the number of levels in any PBS and in the functional packing keyswitch is 6
- the log2 of the base in any PBS and in the functional packing keyswitch is 7
- $\ell_{\mathsf{KS}} = 14$
- $\log_2(\beta_{\mathsf{KS}}) = 1$
- the number of levels of the output GGSW ciphertexts from the circuit bootstrap is 7
- the log2 of the base of the output GGSW ciphertexts from the circuit bootstrap is 4

I.3 Parameter Table for AP#1

We report in Tables 5, 6, 7, 8, 9 the parameters for AP#1.

Algorithm 1: $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{WoP\text{-}PBS}((\mathsf{ct}_0, \ldots, \mathsf{ct}_{\kappa-1}), \mathsf{PUB}, L)$

Context:
$\begin{cases} \Delta_i : \text{ scaling factor for the ciphertext } \mathsf{ct}_i \\ \delta_i : \text{ bits occupied by message in ciphertext } \mathsf{ct}_i \text{ starting from } \Delta_i \\ \Omega = 2^{\sum_{i=0}^{\kappa-1} \delta_i} \\ (\beta_{\mathbf{CB}}, \ell_{\mathbf{CB}}) : \text{ the base and level of the output GGSW ciphertexts to the circuit bootstrapping} \\ (\varkappa, \vartheta) \in \mathbb{N} \times \mathbb{N} \text{ defining the modulus switching in the generalized PBS [CLOT21]} \end{cases}$

Input:
$\begin{cases} (\mathsf{ct}_0, \ldots, \mathsf{ct}_{\kappa-1}) \text{ encrypting } \mathsf{msg} = (m_0, \ldots, m_{\kappa-1}) \\ \text{with for all } 0 \leq i < \kappa, \ \mathsf{Decode}(\mathsf{Decrypt}(\mathsf{ct}_i)) = m_i \\ \mathsf{PUB} : \text{ public keys required for the whole algorithm} \\ L = [l_0, l_1, \cdots, l_{\Omega-1}] : \text{ a LUT, s.t. } l_h \in \mathbb{Z}_\omega \end{cases}$

Output: $\mathsf{ct}_{\mathsf{out}}$ encrypting l_{msg}

for $i \in [\![0; \kappa - 1]\!]$ **do**
  for $j \in [\![0; \delta_i - 2]\!]$ **do**
    `/* Extract from the LSB of the message (use generalized PBS from [CLOT21]) */`
    $\alpha_{i,j} = \frac{\Delta_i \cdot 2^j}{2}$
    $L_{i,j} = [-\alpha_{i,j}, \cdots, -\alpha_{i,j}]$
    $c_i \leftarrow \mathsf{KS\text{-}PBS}(\mathsf{ct}_i + (0, \cdots, 0, \alpha_{i,j}), \mathsf{PUB}, L_{i,j}, (\varkappa = \log_2(\Delta_i) + j, \vartheta = 0))$
    $c'_i \leftarrow c_i - (0, \cdots, 0, \alpha_{i,j})$
    `/* Subtract the extracted bit from the original ciphertext */`
    $\mathsf{ct}_i \leftarrow \mathsf{Sub}(\mathsf{ct}_i, c'_i)$
    `/* Circuit bootstrap [CGGI20] the extracted bit into a GGSW */`
    $\overline{\overline{C}}_{i,j} \leftarrow \mathsf{CircuitBootstrap}(c'_i, \mathsf{PUB}, (\beta_{\mathbf{CB}}, \ell_{\mathbf{CB}}), (\varkappa = \log_2(\Delta_i) + j, \vartheta = 0))$
  `/* Circuit bootstrap [CGGI20] the last bit into a GGSW */`
  $\overline{\overline{C}}_{i,j} \leftarrow \mathsf{CircuitBootstrap}(\mathsf{ct}_i, \mathsf{PUB}, (\beta_{\mathbf{CB}}, \ell_{\mathbf{CB}}), (\varkappa = \log_2(\Delta_i) + \delta_i - 1, \vartheta = 0))$
`/* Vertical Packing LUT evaluation [CGGI20] */`
$\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{VPLut}\left(\left\{\overline{\overline{C}}_{i,j}\right\}_{i \in [\![0;\kappa-1]\!]}^{j \in [\![0;\delta_i-1]\!]}, L\right)$
return $\mathsf{ct}_{\mathsf{out}}$

Algorithm 2: $(\mathsf{ct}_j)_{j\in[\![0,\gamma]\!]} \leftarrow \mathsf{Decomp}\left(\mathsf{ct}_{\mathsf{in}}, \vec{\beta}, \vec{p}, \mathsf{PUB}\right)$

Context:
$$\begin{cases} (q, p, \deg): \text{ parameters of } \mathsf{ct}_{\mathsf{in}} \\ \mu := \deg \cdot (p-1) \\ \gamma := \gamma_{\vec{\beta}}(\mu) \\ \vec{s} \in \mathbb{Z}^n : \text{the secret key} \\ P_{i,\vec{\beta}} : \text{a LUT for } x \to r_{i,\vec{\beta}}(x) \cdot \frac{q}{2 \cdot p_i}, i \in [\![0, \kappa-1]\!] \end{cases}$$

Input:
$$\begin{cases} \mathsf{ct}_{\mathsf{in}} : \text{ LWE encryption of a message } m \\ (\vec{p}, \vec{\beta}) \in \mathbb{N}^{\kappa^2} \\ \mathsf{PUB}\text{: public material for KS-PBS} \end{cases}$$

Output: $(\mathsf{ct}_j)_{j\in[\![0,\gamma]\!]}$ encrypting the message m

for $j \in [\![0, \gamma]\!]$ **do**
  $\mathsf{ct}_j \leftarrow \mathsf{KS\text{-}PBS}(\mathsf{ct}_{\mathsf{in}}, \mathsf{PUB}, P_i)$
  with ct_j LWE encryption with parameters $\left(q, \beta_j, p_j, \deg = \min(\frac{\beta_j - 1}{p_j - 1}, \frac{q_{j-1,\vec{\beta}}(\mu)}{p_j - 1})\right)$
end
return $(\mathsf{ct}_j)_{j\in[\![0,\gamma]\!]}$

Algorithm 4: $(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}) \leftarrow \mathsf{ModReduction}_1((\mathsf{ct}_0, \ldots, \mathsf{ct}_{\kappa-1}), \mathsf{PUB})$

Context:
$$\begin{cases} P_j : \kappa \text{ r-redundant LUT for } x \in \mathbb{Z}_{p_{\kappa-1}} \to \mathsf{Decomp}_j\left(x \cdot \prod_{h=0}^{\kappa-2} \beta_h \mod Q\right) \\ \mathsf{Decomp}_j \text{ is the j-th element in the decomposition in base } (\beta_0, \ldots, \beta_{\kappa-1}) \end{cases}$$

Input:
$$\begin{cases} (\mathsf{ct}_0, \ldots, \mathsf{ct}_{\kappa-1}), \text{ encrypting } \mathsf{msg} = m_0 + \sum_{i=1}^{\kappa-1} m_i \cdot \left(\prod_{j=0}^{i-1} \beta_i\right) \\ \text{s.t. } \mathsf{ct}_i \text{ encrypts message } m_i \text{ with parameters } (\beta_i, p_i) \\ \mathsf{PUB}\text{: public material for KS-PBS} \end{cases}$$

Output: $(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1})$, encrypting $\mathsf{msg} = m_0 + \sum_{i=1}^{\kappa-1} m_i \cdot \left(\prod_{j=0}^{i-1} \beta_i\right) \mod \Omega$

`/* Decompose message in block` $\kappa - 1$ `with respect to base` $(\beta_0, \ldots, \beta_{\kappa-1})$ `*/`
for $j \in [\![0; \kappa-1]\!]$ **do**
  $c_j \leftarrow \mathsf{KS\text{-}PBS}(\mathsf{ct}_{\kappa-1}, \mathsf{PUB}, P_j)$
`/* Add (as in Section 4.2) decomposition to all the blocks up to` $\kappa - 2$ `*/`
for $j \in [\![0; \kappa-2]\!]$ **do**
  $\mathsf{ct}'_j \leftarrow \mathsf{Add}(\mathsf{ct}_j, c_j)$
`/* Replace` $\kappa - 1$ `block with` $\kappa - 1$ `element in decomposition */`
$\mathsf{ct}'_{\kappa-1} \leftarrow c_{\kappa-1}$
return $(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1})$

Algorithm 5: $\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right) \leftarrow \mathsf{ModReduction}_2((\mathsf{ct}_0, \ldots, \mathsf{ct}_{\kappa-1}), \mathsf{PUB})$

Context: $\begin{cases} \vec{\nu} = (\nu_0, \nu_1, \ldots, \nu_{\kappa-1}) \text{ be a convenient decomposition s.t.} \\ \prod_{h=0}^{\kappa-2} \beta_h \mod Q = \nu_0 + \nu_1\beta_0 + \nu_2\beta_0\beta_1 + \ldots + \nu_{\kappa-2}\prod_{j=0}^{\kappa-3}\beta_j \end{cases}$

Input: $\begin{cases} (\mathsf{ct}_0, \ldots, \mathsf{ct}_{\kappa-1}), \text{ encrypting } \mathsf{msg} = m_0 + \sum_{i=1}^{\kappa-1} m_i \left(\prod_{j=0}^{i-1}\beta_i\right) \\ \text{s.t. } \mathsf{ct}_i \text{ encrypts message } m_i \text{ with parameters } (\beta_i, p_i) \end{cases}$

Output: $\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right)$ encrypting $\mathsf{msg} = m_0 + \sum_{i=1}^{\kappa-1} m_i \left(\prod_{j=0}^{i-1}\beta_i\right) \mod \Omega$

`/* Copy input and set the` $\kappa - 1$ `block to zero (trivial encryption) */`

1 $\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right) \leftarrow (\mathsf{ct}_0, \ldots, \mathsf{ct}_{\kappa-2}, 0)$

2 **for** $j \in [\![0; \kappa-2]\!]$ **do**

`/* Multiply block` $\kappa - 1$ `times` ν_j`, Multiplication with a Positive Constant as in Section 4.2 */`

3 **if** $\nu_j < 0$ **then**

4 $c_j \leftarrow \mathsf{ScalarMul}(\mathsf{ct}_{\kappa-1}, -\nu_j)$

5 **else**

6 $c_j \leftarrow \mathsf{ScalarMul}(\mathsf{ct}_{\kappa-1}, \nu_j)$

`/* Decompose (as in Supplementary Material D)` c_j `block starting from the` β_j `*/`

7 $(c_{j,0}, \ldots, c_{j,\kappa-j-1}) \leftarrow \mathsf{Decomp}\left(c_j, (\beta_i)_{i\in[\![j,\kappa-1]\!]}, (p_i)_{i\in[\![j,\kappa-1]\!]}, \mathsf{PUB}\right)$

`/* Pad (as in Supplementary Material D) the carry to fit with the output */`

8 $\left(c'_{j,0}, \ldots, c'_{j,\kappa-1}\right) \leftarrow \mathsf{Pad}\left((c_{j,0}, \ldots, c_{j,\kappa-j-1}), j, (\beta_i)_{i\in[\![0,\kappa-1]\!]}, (p_i)_{i\in[\![0,\kappa-1]\!]}\right)$

`/* Update the output, addition and subtraction as in Section 4.3 */`

9 **if** $\nu_j < 0$ **then**

10 $\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right) \leftarrow \mathsf{Add}\left(\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right), \left(c'_{j,0}, \ldots, c'_{j,k-1}\right)\right)$

11 **else**

12 $\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right) \leftarrow \mathsf{Sub}\left(\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right), \left(c'_{j,0}, \ldots, c'_{j,k-1}\right)\right)$

13 **return** $\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right)$

Addition	Scalar Multiplication	LUT Evaluation	Multiplication
0.3 μs	0.6 μs	2.7 s	5.7 s

Table 1: Benchmarks for homomorphic computation with 16 bits of precision using the CRT approach and the WoP-PBS. Timings are amortized with a trivial parallelism across CRT residues (7,8,9,11 and 13). The parameter set used is under the crypto ID 9.

Parameters					**Radix Operators**		
Message	Carry	Ω	κ	Crypto ID	$\times$	Full Propagate	$+$
2^1	2^1	2^{16}	16	1	3.75 s	0.457 s	5.92 μs
2^2	2^2	2^{16}	8	2	1.81 s	0.457 s	5.26 μs
2^3	2^3	2^{16}	6	3	2.89 s	0.961 s	7.94 μs
2^4	2^4	2^{16}	4	4	8.29 s	3.67 s	26.6 μs
Parameters					**Parallel CRT Operators**		
Message	Carry	Ω	κ	Crypto ID	$\times$	Clean Carries	$+$
$\approx 2^4$	$\approx 2^4$	$\Omega_{\mathsf{CRT},16 \text{ bits}} > 2^{16}$	5	4	0.629 s	0.473 s	2.91 μs

Table 2: Benchmarks for homomorphic additions, full propagate, and additions, for 16 bit homomorphic integers based on radix and CRT approaches. The parallelization is done with κ threads.

Parameters					**Radix Operators**		
Message	Carry	Ω	κ	Crypto ID	$\times$	Full Propagate	$+$
2^1	2^1	2^{32}	32	5	15.6 s	0.957 s	13.3 μs
2^2	2^2	2^{32}	16	6	7.56 s	0.913 s	11.9 μs
2^3	2^3	2^{32}	11	7	9.60 s	1.75 s	16.1 μs
2^4	2^4	2^{32}	8	8	29.7 s	7.71 s	78.1 μs
Parameters					**Parallel CRT Operators**		
Message	Carry	Ω	κ	Crypto ID	$\times$	Clean Carries	$+$
$\approx 2^6$	$\approx 2^2$	$\Omega_{\mathsf{CRT},32 \text{ bits}} > 2^{32}$	6	8	2.21 s	0.473 s	4.28 μs

Table 3: Benchmarks for homomorphic additions, full propagate, and additions, for 32 bit homomorphic integers based on radix and CRT approaches. The parallelization is done with κ threads.

Param Set ID	n	k	N	σ_{LWE}	σ_{GLWE}	β_{PBS}	ℓ_{PBS}	β_{KS}	ℓ_{KS}
1	585	1	2^{10}	9.141776004202573 e-5	2.989040792967434 e-8	2^8	2^2	2^2	2^5
2	720	1	2^{11}	7.747831515176779 e-6	2.2148688116005568 e-16	2^{23}	2^1	2^4	2^5
3	829	1	2^{12}	1.0562341599676662 e-6	2.168404344971009 e-19	2^{23}	2^1	2^2	2^{11}
4	953	1	2^{14}	1.0945755939288365 e-7	2.168404344971009 e-19	2^{15}	2^2	2^4	2^5
5	585	1	2^{10}	9.141776004202573 e-5	2.989040792967434 e-8	2^8	2^2	2^8	2^2
6	720	1	2^{11}	7.747831515176779 e-6	2.2148688116005568 e-16	2^{23}	2^1	2^4	2^5
7	829	1	2^{12}	1.0562341599676662 e-6	2.168404344971009 e-19	2^{23}	2^1	2^2	2^{11}
8	953	1	2^{14}	1.0945755939288365 e-7	2.168404344971009 e-19	2^{15}	2^2	2^4	2^5

Table 4: Optimized parameters for AP type #1 designed for radix or CRT approaches.

$\log_2(p)$	$\log_2(\nu)$	k	$\log_2(N)$	n	$\log_2(\beta_{\mathsf{PBS}})$	$\ell_{\mathbf{PBS}}$	$\log_2(\beta_{\mathsf{KS}})$	ℓ_{KS}	σ^2_{LWE}	σ^2_{GLWE}
1	0	3	9	552	1	19	3	3	10^{-6} . 0.027930383996509764	10^{-22} . 0.06620333228945383
1	1	3	9	552	1	19	3	3	10^{-6} . 0.027930383996509764	10^{-22} . 0.06620333228945383
1	2	3	9	553	1	19	3	3	10^{-6} . 0.026927599620065487	10^{-22} . 0.06620333228945383
1	3	3	9	553	1	19	3	3	10^{-6} . 0.026927599620065487	10^{-22} . 0.06620333228945383
1	4	3	9	555	1	19	3	3	10^{-6} . 0.025028747135636498	10^{-22} . 0.06620333228945383
1	5	3	9	564	1	19	3	3	10^{-6} . 0.018010498061808883	10^{-22} . 0.06620333228945383
1	6	3	9	552	2	12	3	3	10^{-6} . 0.027930383996509764	10^{-22} . 0.06620333228945383
1	7	3	9	552	2	12	3	3	10^{-6} . 0.027930383996509764	10^{-22} . 0.06620333228945383
1	8	3	9	553	2	12	3	3	10^{-6} . 0.026927599620065487	10^{-22} . 0.06620333228945383
1	9	3	9	553	2	12	3	3	10^{-6} . 0.026927599620065487	10^{-22} . 0.06620333228945383
1	10	3	9	556	2	12	3	3	10^{-6} . 0.024130140207139664	10^{-22} . 0.06620333228945383
1	11	3	9	572	2	12	3	3	10^{-6} . 0.013442857673012663	10^{-22} . 0.06620333228945383
1	12	3	9	525	3	9	5	2	10^{-6} . 0.074957841099394 63	10^{-22} . 0.06620333228945383
1	13	3	9	528	3	9	5	2	10^{-6} . 0.067170607140396 3	10^{-22} . 0.06620333228945383
1	14	3	9	540	3	9	5	2	10^{-6} . 0.043313979853044 83	10^{-22} . 0.06620333228945383
1	15	2	10	563	2	16	3	3	10^{-6} . 0.018681209388186886	10^{-28} . 0.00049056438526008 63
1	16	2	10	565	2	16	3	3	10^{-6} . 0.017363867279360456	10^{-28} . 0.00049056438526008 63
1	17	2	10	571	2	16	3	3	10^{-6} . 0.013943469975306 33	10^{-28} . 0.00049056438526008 63
1	18	2	10	567	2	16	5	2	10^{-6} . 0.016139420132290 83	10^{-28} . 0.00049056438526008 63
1	19	2	10	533	3	12	5	2	10^{-6} . 0.055947791444600 59	10^{-28} . 0.00049056438526008 63
1	20	2	10	534	3	12	5	2	10^{-6} . 0.053939098290534 24	10^{-28} . 0.00049056438526008 63
1	21	2	10	540	3	12	5	2	10^{-6} . 0.043313979853044 83	10^{-28} . 0.00049056438526008 63
1	22	2	10	535	4	10	5	2	10^{-6} . 0.052002523233392 7	10^{-28} . 0.00049056438526008 63
1	23	2	10	542	4	10	5	2	10^{-6} . 0.040259609366662636	10^{-28} . 0.00049056438526008 63
1	24	2	10	535	5	8	5	2	10^{-6} . 0.052002523233392 7	10^{-28} . 0.00049056438526008 63
1	25	2	10	546	5	8	5	2	10^{-6} . 0.034781835628854 13	10^{-28} . 0.00049056438526008 63
1	26	2	10	549	6	7	5	2	10^{-6} . 0.031168413902818075	10^{-28} . 0.00049056438526008 63
1	27	2	10	554	7	6	5	2	10^{-6} . 0.025960818205037 99	10^{-28} . 0.00049056438526008 63
1	28	2	10	564	9	5	5	2	10^{-6} . 0.018010498061808 83	10^{-28} . 0.00049056438526008 63
1	29	2	10	582	11	4	6	2	10^{-6} . 0.009326075940779933	10^{-28} . 0.00049056438526008 63
1	30	2	10	540	22	2	10	1	10^{-6} . 0.043313979853044 83	10^{-28} . 0.00049056438526008 63
1	31	2	10	577	46	1	11	1	10^{-6} . 0.011196834888467916	10^{-28} . 0.00049056438526008 63
2	0	3	9	622	1	19	3	4	10^{-6} . 0.002160370983179862	10^{-22} . 0.06620333228945383

Table 5: Optimal parameters for AP type #1.

$\log_2(p)$	$\log_2(\nu)$	k	$\log_2(N)$	n	$\log_2(\beta_{\mathsf{PBS}})$	$\ell_{\mathbf{PBS}}$	$\log_2(\beta_{\mathsf{KS}})$	ℓ_{KS}	σ^2_{LWE}	σ^2_{GLWE}
2	1	3	9	622	1	19	3	4	10^{-6} . 0.0021603709831179862	10^{-22} . 0.06620333228945383
2	2	3	9	623	1	19	3	4	10^{-6} . 0.0020828071993001362	10^{-22} . 0.06620333228945383
2	3	3	9	625	1	19	3	4	10^{-6} . 0.0019359339658177729	10^{-22} . 0.06620333228945383
2	4	3	9	633	1	19	3	4	10^{-6} . 0.0014449619703755682	10^{-22} . 0.06620333228945383
2	5	3	9	572	2	12	5	2	10^{-6} . 0.0134428576730126663	10^{-22} . 0.06620333228945383
2	6	3	9	572	2	12	5	2	10^{-6} . 0.0134428576730126663	10^{-22} . 0.06620333228945383
2	7	3	9	572	2	12	5	2	10^{-6} . 0.0134428576730126663	10^{-22} . 0.06620333228945383
2	8	3	9	573	2	12	5	2	10^{-6} . 0.0129602188506096696	10^{-22} . 0.06620333228945383
2	9	3	9	576	2	12	5	2	10^{-6} . 0.0116138052550568805	10^{-22} . 0.06620333228945383
2	10	3	9	609	2	12	4	3	10^{-6} . 0.0034750318096968095	10^{-22} . 0.06620333228945383
2	11	3	9	573	3	9	5	2	10^{-6} . 0.0129602188506096696	10^{-22} . 0.06620333228945383
2	12	3	9	576	3	9	5	2	10^{-6} . 0.0116138052550568805	10^{-22} . 0.06620333228945383
2	13	3	9	612	3	9	4	3	10^{-6} . 0.0031140170670071133	10^{-22} . 0.06620333228945383
2	14	2	10	580	2	16	5	2	10^{-6} . 0.0100336160497455333	10^{-28} . 0.00049056438526008630
2	15	2	10	582	2	16	5	2	10^{-6} . 0.0093260759407799330	10^{-28} . 0.00049056438526008630
2	16	2	10	604	2	16	4	3	10^{-6} . 0.0041721038572301150	10^{-28} . 0.00049056438526008630
2	17	2	10	639	2	16	4	3	10^{-6} . 0.0011603281414930846	10^{-28} . 0.00049056438526008630
2	18	2	10	581	3	12	5	2	10^{-6} . 0.0096733792048360970	10^{-28} . 0.00049056438526008630
2	19	2	10	583	3	12	5	2	10^{-6} . 0.0089912419033166820	10^{-28} . 0.00049056438526008630
2	20	2	10	593	3	12	5	2	10^{-6} . 0.0062377365610731700	10^{-28} . 0.00049056438526008630
2	21	2	10	583	4	10	5	2	10^{-6} . 0.0089912419033166820	10^{-28} . 0.00049056438526008630
2	22	2	10	584	4	10	6	2	10^{-6} . 0.0086684294098993590	10^{-28} . 0.00049056438526008630
2	23	2	10	584	5	8	5	2	10^{-6} . 0.0086684294098993590	10^{-28} . 0.00049056438526008630
2	24	2	10	588	5	8	6	2	10^{-6} . 0.0074889918113035480	10^{-28} . 0.00049056438526008630
2	25	2	10	591	6	7	6	2	10^{-6} . 0.0067109740549715220	10^{-28} . 0.00049056438526008630
2	26	2	10	597	7	6	6	2	10^{-6} . 0.0053890221420767350	10^{-28} . 0.00049056438526008630
2	27	2	10	608	9	5	6	2	10^{-6} . 0.0036044420673303845	10^{-28} . 0.00049056438526008630
2	28	2	10	653	11	4	7	2	10^{-9} . 0.6954583264270033	10^{-28} . 0.00049056438526008630
2	29	2	10	594	22	2	6	2	10^{-6} . 0.0060137831501596410	10^{-28} . 0.00049056438526008630
3	0	3	9	676	1	19	3	4	10^{-9} . 0.2999494080391601300	10^{-22} . 0.06620333228945383
3	1	3	9	677	1	19	3	4	10^{-9} . 0.2891803265984545500	10^{-22} . 0.06620333228945383
3	2	3	9	681	1	19	3	4	10^{-9} . 0.2498340812943750	10^{-22} . 0.06620333228945383
3	3	3	9	707	1	19	3	4	10^{-9} . 0.0965585609841147400	10^{-22} . 0.06620333228945383
3	4	3	9	646	2	12	4	3	10^{-9} . 0.8983094496825333	10^{-22} . 0.06620333228945383
3	5	3	9	646	2	12	4	3	10^{-9} . 0.8983094496825333	10^{-22} . 0.06620333228945383
3	6	3	9	646	2	12	4	3	10^{-9} . 0.8983094496825333	10^{-22} . 0.06620333228945383
3	7	3	9	647	2	12	4	3	10^{-9} . 0.8660574519678933	10^{-22} . 0.06620333228945383
3	8	3	9	652	2	12	4	3	10^{-9} . 0.7213572091207377	10^{-22} . 0.06620333228945383
3	9	3	9	701	2	12	5	3	10^{-9} . 0.1202448200185095200	10^{-22} . 0.06620333228945383
3	10	3	9	647	3	9	4	3	10^{-9} . 0.8660574519678933	10^{-22} . 0.06620333228945383
3	11	3	9	653	3	9	4	3	10^{-9} . 0.6954583264270033	10^{-22} . 0.06620333228945383
3	12	2	10	641	2	15	4	3	10^{-6} . 0.0010785053275048299	10^{-28} . 0.00049056438526008630
3	13	2	10	641	2	16	4	3	10^{-6} . 0.0010785053275048299	10^{-28} . 0.00049056438526008630
3	14	2	10	643	2	16	4	3	10^{-6} . 0.0010024524096774556	10^{-28} . 0.00049056438526008630
3	15	2	10	648	2	16	4	3	10^{-9} . 0.8349633974953653	10^{-28} . 0.00049056438526008630
3	16	2	10	675	2	16	7	2	10^{-9} . 0.3111195302906904	10^{-28} . 0.00049056438526008630
3	17	2	10	642	3	12	4	3	10^{-6} . 0.0010397837584840370	10^{-28} . 0.00049056438526008630
3	18	2	10	643	3	12	4	3	10^{-6} . 0.0010024524096774556	10^{-28} . 0.00049056438526008630
3	19	2	10	652	3	12	4	3	10^{-9} . 0.7213572091207377	10^{-28} . 0.00049056438526008630
3	20	2	10	619	4	10	6	2	10^{-6} . 0.0024108274700341565	10^{-28} . 0.00049056438526008630
3	21	2	10	630	4	10	6	2	10^{-6} . 0.0016124795409948046	10^{-28} . 0.00049056438526008630

Table 6: Optimal parameters for AP type #1.

$\log_2(p)$	$\log_2(\nu)$	k	$\log_2(N)$	n	$\log_2(\beta_{\mathsf{PBS}})$	$\ell_{\mathbf{PBS}}$	$\log_2(\beta_{\mathsf{KS}})$	ℓ_{KS}	σ^2_{LWE}	σ^2_{GLWE}
3	22	2	10	620	5	8	6	2	10^{-6} . 0.00232427154870724445	10^{-28} . 0.00049056438526008 63
3	23	2	10	636	5	8	6	2	10^{-6} . 0.00129484749589061 67	10^{-28} . 0.00049056438526008 63
3	24	2	10	641	6	7	6	2	10^{-6} . 0.00107850532750482 99	10^{-28} . 0.00049056438526008 63
3	25	2	10	652	7	6	6	2	10^{-9} . 0.72135720912073 77	10^{-28} . 0.00049056438526008 63
3	26	2	10	668	9	5	7	2	10^{-9} . 0.40186680269443 85	10^{-28} . 0.00049056438526008 63
3	27	2	10	664	14	3	7	2	10^{-9} . 0.46515660573676 25	10^{-28} . 0.00049056438526008 63
3	28	2	10	644	22	2	7	2	10^{-9} . 0.96646136801872 8	10^{-28} . 0.00049056438526008 63
4	0	2	10	750	1	22	3	4	10^{-9} . 0.02004390690003296	10^{-28} . 0.00049056438526008 63
4	1	2	10	750	1	22	3	4	10^{-9} . 0.02004390690003296	10^{-28} . 0.00049056438526008 63
4	2	2	10	750	1	22	3	4	10^{-9} . 0.02004390690003296	10^{-28} . 0.00049056438526008 63
4	3	2	10	750	1	23	3	4	10^{-9} . 0.01932427065468256 5	10^{-28} . 0.00049056438526008 63
4	4	2	10	751	1	23	3	4	10^{-9} . 0.05579583808395868	10^{-28} . 0.00049056438526008 63
4	5	2	10	722	1	24	4	3	10^{-9} . 0.11176552076720861	10^{-28} . 0.00049056438526008 63
4	6	2	10	703	1	24	5	3	10^{-9} . 0.06458309689518306	10^{-28} . 0.00049056438526008 63
4	7	1	11	718	1	23	5	3	10^{-9} . 0.14436526192689 29	10^{-28} . 0.00049056438526008 63
4	8	2	10	696	2	15	5	3	10^{-9} . 0.14436526192689 29	10^{-28} . 0.00049056438526008 63
4	9	2	10	696	2	15	5	3	10^{-9} . 0.14436526192689 29	10^{-28} . 0.00049056438526008 63
4	10	2	10	696	2	15	5	3	10^{-9} . 0.14436526192689 29	10^{-28} . 0.00049056438526008 63
4	11	2	10	696	2	16	5	3	10^{-9} . 0.14436526192689 29	10^{-28} . 0.00049056438526008 63
4	12	2	10	697	2	16	5	3	10^{-9} . 0.13918211696567154	10^{-28} . 0.00049056438526008 63
4	13	2	10	699	2	16	5	3	10^{-9} . 0.12936741708920493	10^{-28} . 0.00049056438526008 63
4	14	2	10	710	2	16	5	3	10^{-9} . 0.08652726747166614	10^{-28} . 0.00049056438526008 63
4	15	2	10	696	3	12	5	3	10^{-9} . 0.14436526192689 29	10^{-28} . 0.00049056438526008 63
4	16	2	10	697	3	12	5	3	10^{-9} . 0.13918211696567154	10^{-28} . 0.00049056438526008 63
4	17	2	10	700	3	12	5	3	10^{-9} . 0.12472273964338219	10^{-28} . 0.00049056438526008 63
4	18	2	10	719	3	12	5	3	10^{-9} . 0.06226437043159755	10^{-28} . 0.00049056438526008 63
4	19	2	10	674	4	10	7	2	10^{-9} . 0.32270562814671 79	10^{-28} . 0.00049056438526008 63
4	20	2	10	700	4	10	7	2	10^{-9} . 0.12472273964338219	10^{-28} . 0.00049056438526008 63
4	21	2	10	676	5	8	7	2	10^{-9} . 0.29994940803916013	10^{-28} . 0.00049056438526008 63
4	22	2	10	677	6	7	7	2	10^{-9} . 0.28918032659845455	10^{-28} . 0.00049056438526008 63
4	23	2	10	679	7	6	7	2	10^{-9} . 0.26878820886365 64	10^{-28} . 0.00049056438526008 63
4	24	2	10	682	9	5	7	2	10^{-9} . 0.24086429006954 38	10^{-28} . 0.00049056438526008 63
4	25	2	10	687	11	4	7	2	10^{-9} . 0.20062086142970575	10^{-28} . 0.00049056438526008 63
4	26	2	10	698	15	3	7	2	10^{-9} . 0.13418506242073496	10^{-28} . 0.00049056438526008 63
4	27	2	10	688	44	1	14	1	10^{-9} . 0.19341797208391668	10^{-28} . 0.00049056438526008 63
5	0	1	11	736	1	22	5	3	10^{-9} . 0.03344198833748049	10^{-28} . 0.00049056438526008 63
5	1	1	11	736	1	22	5	3	10^{-9} . 0.03344198833748049	10^{-28} . 0.00049056438526008 63
5	2	1	11	736	1	23	5	3	10^{-9} . 0.03344198833748049	10^{-28} . 0.00049056438526008 63
5	3	1	11	737	1	23	5	3	10^{-9} . 0.03224132087059 0756	10^{-28} . 0.00049056438526008 63
5	4	1	11	739	1	23	5	3	10^{-9} . 0.02996776091285743	10^{-28} . 0.00049056438526008 63
5	5	1	11	748	1	23	5	3	10^{-9} . 0.02156457520277885 7	10^{-28} . 0.00049056438526008 63
5	6	1	11	736	2	15	5	3	10^{-9} . 0.03344198833748049	10^{-28} . 0.00049056438526008 63
5	7	1	11	736	2	15	5	3	10^{-9} . 0.03344198833748049	10^{-28} . 0.00049056438526008 63
5	8	1	11	736	2	15	5	3	10^{-9} . 0.03344198833748049	10^{-28} . 0.00049056438526008 63
5	9	1	11	736	2	15	5	3	10^{-9} . 0.03344198833748049	10^{-28} . 0.00049056438526008 63
5	10	1	11	737	2	15	5	3	10^{-9} . 0.03224132087059 0756	10^{-28} . 0.00049056438526008 63
5	11	1	11	738	2	15	5	3	10^{-9} . 0.03108376095913402	10^{-28} . 0.00049056438526008 63
5	12	1	11	742	2	16	5	3	10^{-9} . 0.02685446646819797	10^{-28} . 0.00049056438526008 63

Table 7: Optimal parameters for AP type #1.

$\log_2(p)$	$\log_2(\nu)$	k	$\log_2(N)$	n	$\log_2(\beta_{\mathsf{PBS}})$	$\ell_{\mathbf{PBS}}$	$\log_2(\beta_{\mathsf{KS}})$	ℓ_{KS}	σ^2_{LWE}	σ^2_{GLWE}
5	13	1	11	789	2	16	5	3	10^{-9} . 0.0048160504768999806	10^{-28} . 0.00049056438526008630
5	14	1	11	737	3	12	5	3	10^{-9} . 0.0322413208705907560	10^{-28} . 0.00049056438526008630
5	15	1	11	738	3	12	5	3	10^{-9} . 0.0310837609591340200	10^{-28} . 0.00049056438526008630
5	16	1	11	745	3	12	5	3	10^{-9} . 0.0240646031090244900	10^{-28} . 0.00049056438526008630
5	17	1	11	715	4	9	7	2	10^{-9} . 0.0720703551876124500	10^{-28} . 0.00049056438526008630
5	18	1	11	725	4	9	7	2	10^{-9} . 0.0499993098125263170	10^{-28} . 0.00049056438526008630
5	19	1	11	716	5	8	7	2	10^{-9} . 0.0694828137433744800	10^{-28} . 0.00049056438526008630
5	20	1	11	732	5	8	7	2	10^{-9} . 0.0387087504612288240	10^{-28} . 0.00049056438526008630
5	21	1	11	738	6	7	7	2	10^{-9} . 0.0310837609591340200	10^{-28} . 0.00049056438526008630
5	22	1	11	737	7	6	8	2	10^{-9} . 0.0322413208705907560	10^{-28} . 0.00049056438526008630
5	23	1	11	754	9	5	8	2	10^{-9} . 0.0173167072768168030	10^{-28} . 0.00049056438526008630
5	24	1	11	732	14	3	8	2	10^{-9} . 0.0387087504612288240	10^{-28} . 0.00049056438526008630
5	25	1	11	732	22	2	8	2	10^{-9} . 0.0387087504612288240	10^{-28} . 0.00049056438526008630
6	0	1	12	829	1	22	4	4	10^{-9} . 0.0011156360606826016	10^{-28} . 0.00000000047019770…
6	1	1	12	830	1	22	4	4	10^{-9} . 0.0010755761232457746	10^{-28} . 0.00000000047019770…
6	2	1	12	831	1	23	4	4	10^{-9} . 0.0010369597214244384	10^{-28} . 0.00000000047019770…
6	3	1	12	839	1	23	4	4	10^{-9} . 0.0007739764830443337	10^{-28} . 0.00000000047019770…
6	4	1	12	806	2	15	5	3	10^{-9} . 0.0025866848530675537	10^{-28} . 0.00000000047019770…
6	5	1	12	806	2	15	5	3	10^{-9} . 0.0025866848530675537	10^{-28} . 0.00000000047019770…
6	6	1	12	806	2	15	5	3	10^{-9} . 0.0025866848530675537	10^{-28} . 0.00000000047019770…
6	7	1	12	806	2	15	5	3	10^{-9} . 0.0025866848530675537	10^{-28} . 0.00000000047019770…
6	8	1	12	807	2	15	5	3	10^{-9} . 0.0024938151253817190	10^{-28} . 0.00000000047019770…
6	9	1	12	809	2	15	5	3	10^{-9} . 0.0023179588621158615	10^{-28} . 0.00000000047019770…
6	10	1	12	817	2	15	5	3	10^{-9} . 0.0017301015756691659	10^{-28} . 0.00000000047019770…
6	11	1	12	821	2	15	9	2	10^{-9} . 0.0014947017412545129	10^{-28} . 0.00000000047019770…
6	12	1	12	807	3	12	5	3	10^{-9} . 0.0024938151253817190	10^{-28} . 0.00000000047019770…
6	13	1	12	810	3	12	5	3	10^{-9} . 0.0022347372017514850	10^{-28} . 0.00000000047019770…
6	14	1	12	785	3	12	8	2	10^{-9} . 0.0055745278730947850	10^{-28} . 0.00000000047019770…
6	15	1	12	775	4	9	8	2	10^{-9} . 0.0080352749932666980	10^{-28} . 0.00000000047019770…
6	16	1	12	781	4	9	8	2	10^{-9} . 0.0064524574974792805	10^{-28} . 0.00000000047019770…
6	17	1	12	777	5	8	8	2	10^{-9} . 0.0074686518221071350	10^{-28} . 0.00000000047019770…
6	18	1	12	799	5	8	8	2	10^{-9} . 0.0033411684907120780	10^{-28} . 0.00000000047019770…
6	19	1	12	810	6	7	8	2	10^{-9} . 0.0022347372017514850	10^{-28} . 0.00000000047019770…
6	20	1	12	821	7	6	9	2	10^{-9} . 0.0014947017412545129	10^{-28} . 0.00000000047019770…
6	21	1	12	784	11	4	8	2	10^{-9} . 0.0057821234082578660	10^{-28} . 0.00000000047019770…
6	22	1	12	796	14	3	8	2	10^{-9} . 0.0037285173898999873	10^{-28} . 0.00000000047019770…
6	23	1	12	803	22	2	9	2	10^{-9} . 0.0028865648301374824	10^{-28} . 0.00000000047019770…
7	0	1	13	861	1	23	6	3	10^{-9} . 0.0003462453330660543300	10^{-28} . 0.00000000047019770…
7	1	1	13	868	1	23	6	3	10^{-9} . 0.0002680581841283959	10^{-28} . 0.00000000047019770…
7	2	1	13	860	2	15	6	3	10^{-9} . 0.0003591395165470425300	10^{-28} . 0.00000000047019770…
7	3	1	13	860	2	15	6	3	10^{-9} . 0.0003591395165470425300	10^{-28} . 0.00000000047019770…
7	4	1	13	860	2	15	6	3	10^{-9} . 0.0003591395165470425300	10^{-28} . 0.00000000047019770…
7	5	1	13	860	2	15	6	3	10^{-9} . 0.0003591395165470425300	10^{-28} . 0.00000000047019770…
7	6	1	13	860	2	15	6	3	10^{-9} . 0.0003591395165470425300	10^{-28} . 0.00000000047019770…
7	7	1	13	861	2	15	6	3	10^{-9} . 0.0003462453330660543300	10^{-28} . 0.00000000047019770…
7	8	1	13	865	2	15	6	3	10^{-9} . 0.0002991347499553079400	10^{-28} . 0.00000000047019770…

Table 8: Optimal parameters for AP type #1.

$\log_2(p)$	$\log_2(\nu)$	k	$\log_2(N)$	n	$\log_2(\beta_{\mathsf{PBS}})$	$\ell_{\mathbf{PBS}}$	$\log_2(\beta_{\mathsf{KS}})$	ℓ_{KS}	σ^2_{LWE}	σ^2_{GLWE}
7	9	1	13	894	2	15	6	3	$10^{-9} \cdot 0.000103602008122216222$	$10^{-28} \cdot 0.000000000004701977\ldots$
7	10	1	13	861	3	11	6	3	$10^{-9} \cdot 0.000346245333066054333$	$10^{-28} \cdot 0.000000000004701977\ldots$
7	11	1	13	863	3	11	6	3	$10^{-9} \cdot 0.000321829164511109036$	$10^{-28} \cdot 0.000000000004701977\ldots$
7	12	1	13	878	3	11	6	3	$10^{-9} \cdot 0.000185967228288651105$	$10^{-28} \cdot 0.000000000004701977\ldots$
7	13	1	13	861	4	9	6	3	$10^{-9} \cdot 0.000346245333066054333$	$10^{-28} \cdot 0.000000000004701977\ldots$
7	14	1	13	868	4	9	6	3	$10^{-9} \cdot 0.000268058184128395959$	$10^{-28} \cdot 0.000000000004701977\ldots$
7	15	1	13	840	5	8	9	2	$10^{-9} \cdot 0.000746188410937163 2$	$10^{-28} \cdot 0.000000000004701977\ldots$
7	16	1	13	880	5	8	9	2	$10^{-9} \cdot 0.000172853384554246 3$	$10^{-28} \cdot 0.000000000004701977\ldots$
7	17	1	13	843	7	6	9	2	$10^{-9} \cdot 0.000668668252295510 2$	$10^{-28} \cdot 0.000000000004701977\ldots$
7	18	1	13	847	8	5	9	2	$10^{-9} \cdot 0.000577688394186414 6$	$10^{-28} \cdot 0.000000000004701977\ldots$
7	19	1	13	872	10	4	9	2	$10^{-9} \cdot 0.000231585844559020 7$	$10^{-28} \cdot 0.000000000004701977\ldots$
7	20	1	13	861	14	3	9	2	$10^{-9} \cdot 0.000346245333066054333$	$10^{-28} \cdot 0.000000000004701977\ldots$
7	21	1	13	879	22	2	19	1	$10^{-9} \cdot 0.000179290448228190 66$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	0	1	14	934	2	15	6	3	$10^{-9} \cdot 0.000023999244008682802$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	1	1	14	934	2	15	6	3	$10^{-9} \cdot 0.000023999244008682802$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	2	1	14	934	2	15	6	3	$10^{-9} \cdot 0.000023999244008682802$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	3	1	14	934	2	15	6	3	$10^{-9} \cdot 0.000023999244008682802$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	4	1	14	934	2	15	6	3	$10^{-9} \cdot 0.000023999244008682802$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	5	1	14	936	2	15	6	3	$10^{-9} \cdot 0.000022306890261359 02$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	6	1	14	957	2	15	5	4	$10^{-9} \cdot 0.000010350812924730065$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	7	1	14	950	2	15	7	3	$10^{-9} \cdot 0.000013369935636477038$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	8	1	14	935	3	11	6	3	$10^{-9} \cdot 0.000023137599323552676$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	9	1	14	925	3	11	7	3	$10^{-9} \cdot 0.000033351160399804 88$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	10	1	14	935	3	11	7	3	$10^{-9} \cdot 0.000023137599323552676$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	11	1	14	924	4	9	7	3	$10^{-9} \cdot 0.000034593158314089 87$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	12	1	14	930	4	9	7	3	$10^{-9} \cdot 0.000027778873020808788$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	13	1	14	904	5	8	10	2	$10^{-9} \cdot 0.000071874613186174 59$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	14	1	14	905	6	7	10	2	$10^{-9} \cdot 0.000069294099465607 19$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	15	1	14	907	7	6	10	2	$10^{-9} \cdot 0.000064407690174730 06$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	16	1	14	912	8	5	10	2	$10^{-9} \cdot 0.000053646500615849 54$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	17	1	14	931	10	4	10	2	$10^{-9} \cdot 0.000026781528342425384$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	18	1	14	938	14	3	10	2	$10^{-9} \cdot 0.000020733876156777412$	$10^{-28} \cdot 0.000000000004701977\ldots$
8	19	1	14	919	42	1	20	1	$10^{-9} \cdot 0.000041532353411342 9$	$10^{-28} \cdot 0.000000000004701977\ldots$

Table 9: Optimal parameters for AP type #1.