Bootstrapping Small Integers With CKKS

Youngjin Bae[1][0000−0001−6870−4504], Jaehyung Kim[1][0000−0002−1624−6326], Damien Stehlé[2][0000−0003−3435−2453], and Elias Suvanto[1,3][0009−0008−2096−4698]

[1] CryptoLab Inc., Seoul, Republic of Korea
{youngjin.bae, elias.suvanto}@cryptolab.co.kr
jaehk@stanford.edu
[2] CryptoLab Inc., Lyon, France
damien.stehle@cryptolab.co.kr
[3] University of Luxembourg, Luxembourg

Abstract. The native plaintexts of the Cheon-Kim-Kim-Song (CKKS) fully homomorphic encryption scheme are vectors of approximations to complex numbers. Drucker *et al* [J. Cryptol.'24] have showed how to use CKKS to efficiently perform computations on bits and small bit-length integers, by relying on their canonical embeddings into the complex plane. For small bit-length integers, Chung *et al* [IACR eprint'24] recently suggested to rather rely on an embedding into complex roots of unity, to gain numerical stability and efficiency. Both works use CKKS in a black-box manner.

Inspired by the design by Bae *et al* [Eurocrypt'24] of a dedicated bootstrapping algorithm for ciphertexts encoding bits, we propose a CKKS bootstrapping algorithm, SI-BTS (small-integer bootstrapping), for ciphertexts encoding small bit-length integers. For this purpose, we build upon the DM/CGGI-to-CKKS conversion algorithm from Boura *et al* [J. Math. Cryptol.'20], to bootstrap canonically embedded integers to integers embedded as roots of unity. SI-BTS allows functional bootstrapping: it can evaluate an arbitrary function of its input while bootstrapping. It may also be used to batch-(functional-)bootstrap multiple DM/CGGI ciphertexts. For example, its amortized cost for evaluating an 8-bit look-up table on 2^{12} DM/CGGI ciphertexts is 3.75ms (single-thread CPU, 128-bit security).

We adapt SI-BTS to simultaneously bootstrap multiple CKKS ciphertexts for bits. The resulting BB-BTS algorithm (batch-bits bootstrapping) allows to decrease the amortized cost of a binary gate evaluation. Compared to Bae *et al*, it gives a 2.4x speed-up.

Keywords: Fully Homomorphic Encryption · Bootstrapping · Binary Circuits · Functional Bootstrapping

1 Introduction

The diverse Fully Homomorphic Encryption (FHE) schemes handle different primary data types. In BGV/BFV [BGV12,Bra12,FV12], a plaintext is a vector of elements in a finite field. DM/CGGI [DM15,CGGI16a] considers bits,

and can be extended to process small bit-length integers [CJP21,KS22]. Finally, CKKS [CKKS17] enables computations on vectors of (approximations to) complex numbers. Even though computations on a second data type can be expressed as computations on a first data type, this incurs a "data type translation" cost. For example, simulating a multiplication between reals with a boolean circuit may incur a large extra cost. For this reason, it can be tempting to choose an FHE scheme whose primary data type matches the considered application, to maximize efficiency. In this work, we go against this intuition, and consider the efficiency of CKKS for computations on small integers, including bits.

Recently, Drucker *et al* [DMPS24] used CKKS to perform computations on vectors of bits, obtaining impressive performance in terms of throughput: thanks to the SIMD nature of CKKS, when the computations to be performed are sufficiently large, the amortized cost of homomorphically evaluating a binary gate becomes very small. The main idea of [DMPS24] is to view the bit $b \in \{0, 1\}$ as a real/complex number and map a vector of such bits to a CKKS plaintext. The latter encoding adds a small noise to the bits. By interpreting binary gates as bivariate polynomials, one can then evaluate binary circuits with CKKS. To handle the noise increase, the authors of [DMPS24] propose to use a noise-cleaning polynomial, which implements the identity function on $\{0, 1\}$ with a vanishing derivative on those points. The vanishing derivatives allow to square the noise, i.e., to double the accuracy. We stress that with CKKS, it is possible to reduce the encryption noise without bootstrapping, since the noise is part of the message. In [ADE+23], following a suggestion from [DMPS24], the authors proposed to use such noise cleaning only after every few gates rather than after every gate, leading to improved throughput. This was used to homomorphically evaluate AES multiple times in parallel. The throughput was further lowered in [BCKS24], which introduced variants of the CKKS bootstrapping algorithm [CHK+18] specifically designed for binary data. Borrowing the figures from [BCKS24, Table 2], evaluating a binary gate with CKKS has an amortized cost of 17.6μs in single-thread CPU (where amortization is over slots and the multiple sequential gates that can be applied between two consecutive bootstraps). For sufficiently large computations, this compares favorably to [DM15,CGGI16a], which typically consumes around 10ms per binary gate [CGGI16b]. Interestingly, the bootstrapping algorithms from [BCKS24] are compatible with DM/CGGI ciphertexts. By relying on fast ring packing [BCK+23], one then obtains a CKKS-based DM/CGGI bootstrapping algorithm for multiple ciphertexts which outperforms other DM/CGGI bootstrapping when the number of ciphertexts to be bootstrapped is around 200.

Drucker *et al* [DMPS24] also considered viewing small bit-length integers as real/complex numbers and using CKKS to perform SIMD homomorphic computations on integers. The cost increases fast with the bit-length, notably because of the considered noise-cleaning strategy. The integer is homomorphically decomposed in base 2 by repeatedly computing the most significant bit. The latter is quite costly as this discontinuous function is implemented using a precise polynomial approximation of a step function. The bits are then cleaned individually before being recombined in a cleaned integer. In [CKKL24], Chung *et al* con-

sidered a different path for enabling small bit-length integer computations with CKKS. Instead of embedding an integer $m \in [0, t)$ for some small t into $\mathbb{C}$ using the identity function, they exploit t-th roots of unity and send m to $\exp(2i\pi m/t)$. By restricting the CKKS plaintext space to small balls around these t points, discrete computations can be performed via numerically stable polynomial interpolations. Indeed, Lagrange's interpolation on equispaced points of the real line suffers from huge oscillations, which is known as Runge's phenomenon. On the contrary, roots of unity make the interpolating polynomial nicely converge to the target function (assuming it is sufficiently smooth). Similarly to binary circuits [DMPS24], the noise grows with homomorphic operations and the data points progressively become less separated. This is also handled with a noise-cleaning polynomial evaluation.

The works above on small integers use CKKS bootstrapping in a black-box manner. This raises the following questions: As in the case of bits, can CKKS bootstrapping be adapted for small bit-length integers? Can we obtain a CKKS-based batch-bootstrapping algorithm for multiple DM/CGGI ciphertexts for small integers? Similarly to DM/CGGI functional/programmable bootstrapping [CJP21,KS22], can we bootstrap and evaluate a function simultaneously?

Contributions. We introduce two new CKKS bootstrapping algorithms for plaintexts respectively encoding small bit-length integers and bits.

The first algorithm, SI-BTS (for small integer bootstrapping), bootstraps ciphertexts whose plaintexts are integers of small bit-sizes (e.g., 8 bits). It can be combined with an arbitrary table look-up, at no extra cost, providing a CKKS analogue to functional bootstrapping [CJP21,KS22] in the context of the DM/CGGI fully homomorphic encryption scheme. Like in [CIM19], several functions of the same plaintexts can be evaluated for a cost that is significantly less than that of applying the functional bootstrap multiple times. Finally, as the inputs and outputs of SI-BTS are compatible with DM/CGGI ciphertexts, SI-BTS can be used to perform functional bootstraps on multiple DM/CGGI ciphertexts at once, rather than running the DM/CGGI functional bootstrapping algorithm in parallel on the multiple ciphertexts.

The second algorithm, BB-BTS (for batch-bits bootstrapping), bootstraps in one go multiple CKKS ciphertexts for bits. For a single ciphertext, it essentially corresponds to the BinBoot algorithm from [BCKS24]. As its cost grows slowly with the number of batched ciphertexts (up to some integer bit-length), when several ciphertexts are considered, it leads to a large throughput improvement compared to [BCKS24].

Implementation. We implemented SI-BTS and BB-BTS in the C++ HEaaN library [Cry22]. For Int-BTS, we designed parameter sets optimizing latency, primarily focusing on obtaining an efficient algorithm for batch functional bootstrapping of DM/CGGI ciphertexts. As showed by its multiple uses (see, e.g., [CJP21,CHMS22,TCBS23]) the importance of functional bootstrapping cannot be overstated. For BB-BTS, we designed parameter sets optimizing throughput.

Table 1 illustrates the performance of CKKS-based functional bootstrapping for integers of various bit-sizes. We compare the performance to the functional

bootstrapping algorithm of DM/CGGI [KS22] and to the BFV/BGV-based functional bootstrapping algorithms of [LW23]. The DM/CGGI figures are retrieved from the tfhe-rs benchmarks page, and correspond to a similar computing environment (single-thread CPU).[4]

	Number of input LWE ciphertexts	Number of input/output bits	Total time	Amortized time
[Zam24]	1	2	6.4ms	6.4ms
		4	12.9ms	12.9ms
		6	104ms	104ms
		8	489ms	489ms
[KS22]	1	8	21s	21s
[LW23]	2^{15}	9	220s	6.7ms
This work	2^{12}	2	3.20s	0.78ms
		4	6.41s	1.57ms
		6	11.0s	2.67ms
		8	15.4s	3.75ms
		10	50.3s	12.3ms

Table 1. Comparison for look-up table evaluations for various input bit-sizes. The figures for [KS22] only provide 100-bit security, while all others aim at 128-bit security.

Table 2 focuses on throughput for binary gate evaluations. The run-time figures for [LMSS23] and [LW24] are borrowed from the corresponding papers. The figures for [BCKS24] and [DMPS24] are borrowed from [BCKS24], and we included only the 'improved version' figures for [DMPS24]. All the experiments were on similar computing environments (single-thread CPU). Our batch bits bootstrapping (BB-BTS) that bootstraps 5 ciphertexts in parallel gives 2.38x faster amortized gate evaluation time than the state-of-the-art [BCKS24]. In this figure, we considered $k = 5$ batched ciphertexts in parallel because it shows the best performance, as illustrated in Table 7. As we increase the batch number k, the amortized bootstrapping time decreases at first but it starts to increase at some point. This is because the cost of BB-BTS depends on k and the benefit of simultaneously computing several bootstrappings is offset by the increased cost. For a more detailed analysis on the effect of k, we refer to Section 6.2.

1.1 Technical Overview

Modulus consumption in bootstrapping. To explain our contributions, we first highlight what makes conventional bootstrapping costly and how this is

[4] Choosing the number of input LWE ciphertexts to be as large as 2^{12} is somewhat necessary to reduce the amortized bootstrapping time. If one uses a smaller number of input LWE ciphertexts instead, one may consider CKKS bootstrapping for fewer slots (i.e., thin bootstrapping). However, as the overall bootstrapping time scales sublinearly with the number of slots, using fewer slots is less efficient in terms of amortized bootstrapping time.

	Number of plaintext slots	Amortized bootstrapping time per gate
[LMSS23]	1	6.49ms
[LW24]	2^{16}	1.5ms
[DMPS24]		27.7μs
[BCKS24]		17.6μs
This work	$5 \cdot 2^{16}$	7.39μs

Table 2. Throughput comparison for BB-BTS. Here number of plaintext slots refers to the number of bits being bootstrapped per single (batched) bootstrapping. All figures correspond to parameters aiming at 128-bit security.

handled with the bootstrapping algorithm from [BCKS24] for bits. CKKS performs plaintext operations on $\mathbb{C}^{N/2}$ by manipulating ciphertexts belonging in R_q^2 where $R_q = \mathbb{Z}[X]/(X^N + 1)$ for some power-of-two integer N. The modulus q may vary but, for any given N, it is bounded from above as else the underlying hard problem, a variant of Ring-LWE [SSTX09,LPR10], does not provide sufficient security. The primary homomorphic operations are component-wise addition, multiplication and complex conjugation, as well as cyclic rotations of the vector coefficients. While many additions, conjugations and rotations can be performed without significant difficulties, repeated multiplications are more difficult to support. They involve a rescaling operation that decreases the modulus q of the ciphertext by a number of bits that grows linearly with the precision of the plaintexts considered in the computations. Therefore, the current modulus directly limits the number of sequential multiplications that can be subsequently performed, if one is restricted to the primary operations mentioned above. The CKKS bootstrapping procedure [CHK$^+$18] takes as input a low-modulus ciphertext and outputs a high-modulus ciphertext that decrypts to the same message, up to some noise. Homomorphic computations can then be run endlessly. Despite many improvements [CCS19,HK20,LLL$^+$21,BTPH22,KPK$^+$22,LLK$^+$22] (among others), CKKS bootstrapping still suffers from two main drawbacks: first, its runtime is high; second, it itself requires significant multiplicative depth and hence consumes a large amount of modulus. Modulus consumption in bootstrapping is a main factor in the efficiency of CKKS: a lower modulus consumption in bootstrapping provides more room for useful computations, helping for throughput; it may also allow to choose a smaller N, which helps for latency.

When it comes to modulus consumption, the two main components of bootstrapping are a linear evaluation phase called CtS (for coefficients to slots), and a non-linear evaluation phase called EvalMod (for evaluation of modular reduction). The other linear phase called StC consumes less modulus, and the remaining bootstrapping component, ModRaise, creates modulus. The input of CtS is a ciphertext whose underlying plaintext is $\mathbf{V} \cdot (\mathbf{x} + \mathbf{I})$, where $\mathbf{V}$ is a matrix that is closely related to the discrete Fourier transform, $\mathbf{x}$ is a vector containing the message and satisfying $\|\mathbf{x}\|_\infty < 1/2$, and $\mathbf{I}$ is a vector whose coordinates are bounded integers. The main task of bootstrapping is to remove $\mathbf{I}$. CtS is a

homomorphic multiplication by $\mathbf{V}^{-1}$: its output is a ciphertext that decrypts to $\approx \mathbf{x} + \mathbf{I}$. EvalMod evaluates, on all coordinates in parallel, a polynomial that approximates the function $x + I \mapsto x$ on a relevant domain. The modulus consumption is driven by two aspects. First, all computations are performed in some precision that is larger than the bit-size of the manipulated data, i.e., $x + I$ with an accuracy that provides enough meaningful bits of the message encoded in x. Second, this per-level modulus consumption is multiplied by the multiplicative depth of CtS and EvalMod.

The first aspect above explains why the BinBoot bootstrapping algorithm from [BCKS24] consumes little modulus and could even be implemented with ring degree as low as $N = 2^{14}$. BinBoot was designed for bootstrapping plaintexts corresponding to bits, encoded into reals as proposed in [DMPS24]. A bit $b \in \{0, 1\}$ is represented by a real $b + \varepsilon$ for some ε satisfying $|\varepsilon| \ll 1$. An essential aspect of BinBoot is that the bit b is encoded into x as $x \approx b/2$. This is in sharp contrast to using an x that satisfies $|x| \ll 1$ as is the case in most other CKKS bootstrapping algorithms. As a result, a small precision suffices for bootstrapping computations: one only needs to handle I and a few more bits to capture a good estimate of b. Further, the multiplicative depth of EvalMod is itself limited as the manipulated data has low bit-size.

As a minor contribution, we note that the cleaning strategy from [BCKS24] can be modified to lower bootstrapping modulus consumption. Error cleaning increases the precision of the bit b, i.e., reduces the magnitude of ε in $b + \varepsilon$. In [BCKS24], cleaning is performed before bootstrapping, and the plaintext is represented on sufficiently many bits to capture this accuracy. Instead, one can perform bootstrapping with a lower precision and clean the error after bootstrapping. Consistently, the accuracy of the plaintext can be decreased. This saves only a few bits of modulus per multiplication level, but this saving is multiplied by the multiplicative depth of bootstrapping.

Bootstrapping integers with low modulus consumption. To minimize modulus consumption in bootstrapping, we would like that the pre-CtS plaintext $\mathbf{V} \cdot (\mathbf{x} + \mathbf{I})$ is such that $\mathbf{x}$ contains m in its most significant bits, as in [BCKS24]. We have two embeddings of integers $m \in [0, t)$ into complex numbers at hand: either encode m as $m \in \mathbb{C}$ (up to some noise) or as $\exp(2i\pi m/t) \in \mathbb{C}$ (up to some noise). CtS works with both encodings. Oppositely, the subsequent bootstrapping step should remove I using a polynomial evaluation, which can be more or less difficult depending on the choice of encoding and scaling. Let us examine the four possibilities at hand:

- map $m/t + I$ to m/t (for all $m \in \mathbb{Z} \cap [0, t)$ and all integer I in some range);
- map $m/t + I$ to $\exp(2i\pi m/t)$;
- map $\exp(2i\pi m/t) + I$ to m/t (for all $m \in \mathbb{Z} \cap [0, t)$ and all Gaussian integer $I = I_1 + iI_2$ with I_1 and I_2 in some range);
- map $\exp(2i\pi m/t) + I$ to $\exp(2i\pi m/t)$.

In principle, any of these can be handled by using a polynomial approximation around the distinguished points (interpolation may not suffice, as the distin-

guished points are noisy). For a growing value of t, the first interpolation task converges to finding a polynomial approximation of the $y \mapsto y \bmod 1$ function. For the remaining three, let us only consider the real part, to simplify: the second function is $y \mapsto \cos y$, the third is $y \mapsto \arccos(y \bmod 1)$ and the fourth is $y \mapsto \arccos(\cos y)$. Out of these, only the second one is differentiable, making it more suitable for polynomial interpolation. We hence choose the second mapping. Conveniently, the usual EvalMod phase of CKKS bootstrapping is typically implemented by a polynomial approximation to a trigonometric function, so that it can be readily replaced by an "EvalExp" that sends $m/t + I$ to

$$\exp\left(2i\pi\left(\frac{m}{t} + I\right)\right) = \exp\left(2i\pi\frac{m}{t}\right) \quad .$$

Indeed, the complex exponential can be computed with a cosine evaluation for the real part and a sine evaluation for the complex part. Further, these can be evaluated efficiently by relying on the double-angle formula. Overall, the resulting bootstrapping, which we refer to as IntRootBoot, sends a ciphertext that contains the integer m embedded as $m/t \in \mathbb{C}$ to a root of unity $\exp(2i\pi m/t) \in \mathbb{C}$.

Interestingly, using EvalExp to bootstrap integers stored in the most significant bits has already been considered in [BGGJ20], in the context of converting DM/CGGI ciphertexts to CKKS ciphertexts. Also, by taking $t = 2$ and considering only the real part, one recovers the BinBoot algorithm from [BCKS24].

An improved tool-box for roots of unity. To obtain good efficiency, we first extend the toolbox for homomorphically manipulating t-th roots of unity. First, we revisit the analysis of the IntRootBoot algorithm from [BGGJ20]. We focus on modulus consumption. By using the sparse-secret encapsulation technique from [BTPH22], one can ensure that $|I| \leq 15$ with probability extremely close to 1, so that $5+\log t$ bits of precision suffice to represent $I+m/t$. A few extra bits are needed to separate the data points, and a little over $(\log N)/2$ bits should be added to handle the inherent inaccuracy of CKKS homomorphic computations. For moderate values of t and $N = 2^{16}$, this adds up to as low as ≈ 30 bits of precision for the elementary CKKS operations, whereas other CKKS bootstrapping algorithms often consider up to 45 or 50 bits of precision. Recall that this modulus gain is multiplied by the multiplicative depth of bootstrapping, which is typically over 10.

We also propose improvements for interpolation from complex roots of unity, which was studied in [CKKL24]. Note that such an interpolation seems necessary for FHE based on IntRootBoot. Indeed, the input complex-plane embedding is $m \mapsto m$ whereas the output embedding is $m \mapsto \exp(2i\pi m/t)$. One then needs to convert roots of unity back to integers at some stage, to be able to use IntRootBoot again. As a first remark, we observe that any monomial x^i for $0 \leq i < t$ can be replaced by $\overline{x}^{t-i}$, as we are only interested in t-th roots of unity. As homomorphic conjugation does not consume modulus, this provides a total degree reduction by a factor 2 and hence allows to save one multiplicative depth. Second, we consider the noise-cleaning functionality. A cleaning function

of degree $t+1$ was proposed in [CKKL24]. Instead, we investigate combined interpolation and cleaning. This may be achieved using Hermite interpolation (which extends Lagrange interpolation by imposing that the derivatives of the polynomial vanish at all interpolation points). For t-th roots of unity, we expect Hermite interpolation to provide polynomials of degree $2t$. This would already be interesting compared to [CKKL24], as evaluation and cleaning could be achieved in depth $1+\log t$ instead of $1+2\log t$. We decrease depth consumption further than this by designing bivariate polynomials in x and $\overline{x}$ of degree $t-1$ that properly interpolate and clean (note that replacing x^i by $\overline{x}^{t-i}$ for large values of i in the Hermite interpolation indeed decreases the degree but does not preserve the cleaning functionality). This also saves one multiplication depth, down to $\log t$.

New bootstrapping algorithms. The design rationale of our first bootstrapping algorithm, SI-BTS, starts from the observation that BinBoot is wasteful. For a single bit of interest, a multiplication consumes more than 20 bits of modulus. For this consumption, we may as well consider small integers rather than bits: the required CKKS precision is about the same, but much more data is handled. SI-BTS hence considers an input ciphertext whose underlying plaintext is a vector $\mathbf{m} \in \mathbb{Z}^{N/2}$ with each coefficient in $[0,t)$ for some integer t. It then calls IntRootBoot, to obtain a high-modulus ciphertext that decrypts to $\exp(2i\pi\mathbf{m}/t)$. If t remains small, the multiplicative depth of the approximate complex exponential evaluation does not grow, as we rely on good polynomial approximations to the sine and cosine functions on a whole interval containing $2 \cdot 15 = 30$ periods, even for $t = 2$. Once we have $\exp(2i\pi\mathbf{m}/t)$, we could keep this format and perform computations on roots of unity as described in [CKKL24]. However, at some stage, we should switch back to integers to apply SI-BTS again. This is achieved by a polynomial interpolation on the roots of unity. Further, we propose to combine the first interpolation after the exponential evaluation with cleaning: this consumes only one extra multiplicative depth compared to a simple interpolation but allows to lower the precision inside bootstrapping (as the output ciphertext is subsequently cleaned). As the approach allows any interpolation from complex roots of unity to integers, we can simultaneously evaluate an arbitrary function from $[0,t)$ to $[0,t)$ at no extra cost. Finally, we note that the input format of SI-BTS closely matches that of DM/CGGI ciphertexts and use it to provide a CKKS-based batch functional bootstrapping algorithm for DM/CGGI.

Our second algorithm, BB-BTS, takes as input many ciphertexts for bits and bootstraps them in a batch. More concretely, assume we have k input CKKS ciphertexts, each of which encrypts a vector $\mathbf{b}_j \in \{0,1\}^{N/2}$. We could execute Binboot k times in parallel. Alternatively, we pack the data into a single ciphertext by homomorphically performing a linear combination $\sum_{0\le j<k} 2^j\mathbf{b}_j$, resulting into a vector whose coefficients are integers in $[0,2^k)$. We then use IntRootBoot to obtain a high-modulus ciphertext that decrypts to the vector of roots of unity $\exp(2i\pi(\sum_j 2^{j-k}b_j))$ (for each slot). To recover individual high-modulus ciphertexts that encrypt the $\mathbf{b}_j$'s, we perform k polynomial interpola-

tions $\exp(2i\pi(\sum_j 2^{j-k}b_j)) \mapsto b_j$. Due to the recursive structure of roots of unity of power-of-two orders, these interpolations are far from arbitrary, allowing for an efficient implementation even for moderate k. Finally, we clean the error terms of the individual bits in parallel. This is far thriftier than cleaning the roots of unity and then converting to integers.

2 Background on CKKS

For a power-of-two $N \geq 1$ and $q \geq 2$, we define the rings $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ and $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$. We let i denote the complex imaginary unit. We let $\overline{x}$ denote the complex conjugate of $x \in \mathbb{C}$. Vectors are denoted in bold lower-case. The notation log refers to base-2 logarithm.

2.1 Encodings

We define the Discrete Fourier Transform (DFT) $\mathsf{DFT} : \mathcal{R} \to \mathbb{C}^{N/2}$ as

$$\mathsf{DFT}(p(X)) = (p(\zeta_j))_{0 \leq j < N/2} \ ,$$

where $\zeta_j = \zeta^{5^j}$ for a complex $2N$-th root of unity $\zeta \in \mathbb{C}$. Its inverse $\mathsf{iDFT} : \mathbb{C}^{N/2} \to \mathcal{R}$ is the inverse Discrete Fourier Transform (iDFT). In the CKKS scheme, messages are elements of $\mathbb{C}^{N/2}$, up to some accuracy quantified by a scaling factor $\Delta > 0$. To encode a message $\mathbf{z} \in \mathbb{C}^{N/2}$ into a plaintext $\mathsf{pt} \in \mathcal{R}$ with a scaling factor Δ, one uses the encoding map $\mathsf{Ecd} : \mathbb{C}^{N/2} \to \mathcal{R}$ defined as

$$\mathsf{Ecd}(\mathbf{z}) = \lfloor \Delta \cdot \mathsf{iDFT}(\mathbf{z}) \rceil \ .$$

The decoding map is defined as $\mathsf{Dcd}(m) = \mathsf{DFT}(m)/\Delta$.

As put forward in [DMPS24], one may focus on a subset of $\mathbb{C}$ to handle discrete data. For example, a bit $b \in \{0, 1\}$ can be viewed as a complex number $b \in \mathbb{C}$. *Bits encoding* maps a vector $\mathbf{b} \in \{0, 1\}^{N/2}$ to $\mathsf{Ecd}(\mathbf{b}) \in \mathbb{C}^{N/2}$. For integers in $[0, t)$ for some arbitrary $t > 0$, one may extend the latter encoding of bits by using the inclusion $\mathbb{Z} \cap [0, t) \subset \mathbb{C}$: an integer vector $\mathbf{v} \in \mathbb{Z}^{N/2}$ is encoded to a plaintext $\mathsf{Ecd}(\mathbf{v}) \in \mathbb{C}^{N/2}$. We will refer to the latter as *integers encoding*. Another way to embed small integers in the complex plane is to use complex roots of unity. As showed in [CKKL24], such an encoding is advantageous in the context of evaluating look-up tables in CKKS, from the perspective of numerical stability. Let $\phi_t : \mathbb{Z}_t \to \mathbb{C}$ denote the map $m \mapsto e^{2\pi i \cdot m/t}$. The *roots-of-unity encoding* of a vector $\mathbf{v} \in \mathbb{Z}_t^{N/2}$ is $\mathsf{Enc} \circ \phi_t^{N/2}$, where $\phi_t^{N/2}$ is the evaluation of ϕ_t on all coefficients of $\mathbf{v}$ in parallel.

CKKS computations induce error growth. If we start with good approximations to encodings as above (for bits, integers or roots-of-unity), computations may lead to less precise approximations. In order to keep the plaintexts close to the expected encodings, it was suggested in [DMPS24] to use cleaning functions to reduce the error. For instance, for bits encoding, one may consider the polynomial $h_1(x) = 3x^2 - 2x^3$ initially introduced in [CKK20]: it has minimal degree

such that $h_1(0) = 0$, $h_1(1) = 1$ and $h_1'(0) = h_1'(1) = 0$. In particular, it satisfies the following cleaning property.

Lemma 1. *For all $b \in \{0,1\}$ and $\varepsilon \in [-1,1]$, we have:*

$$|h_1(b+\varepsilon) - b| \leq 3|\varepsilon|^2 + 2|\varepsilon|^3 \ .$$

Note that $3|\varepsilon|^2 + 2|\varepsilon|^3$ is much smaller than $|\varepsilon|$, if $|\varepsilon|$ is sufficiently small.

In [DMPS24], the authors suggested to clean small integers by extracting bits (with a homomorphic evaluation of binary decomposition), cleaning bits using h_1 and then recombining the cleaned bits. For roots of unity, the authors of [CKKL24] considered the polynomial $((t+1)x - x^{t+1})/t$.

2.2 Ciphertexts and Elementary Operations

A CKKS ciphertext for a message $\mathbf{z} \in \mathbb{C}^{N/2}$ for a secret key $s \in \mathcal{R}$ is a pair $(a,b) \in \mathcal{R}_q$ such that $a \cdot s + b \approx \mathsf{Ecd}(\mathbf{z})$. In that case, we say that the ciphertext encrypts $\mathbf{z}$ in its slots. Sometimes, we have $a \cdot s + b \approx \Delta \cdot z'$ for some scaling factor Δ and where z' starts with the real parts of the coefficients of $\mathbf{z}$ and continues with the imaginary parts. In this case, we say that $\mathbf{z}$ is encrypted in the coefficients.

The homomorphic addition algorithm add takes as input two ciphertexts modulo q and outputs a ciphertext that decrypts to the sum of the plaintexts underlying the input ciphertexts (for encryption with respect to both slots and coefficients). The homomorphic conjugation algorithm conj takes as input a ciphertext modulo q and outputs a ciphertext that decrypts to the coefficient-wise complex conjugate of the plaintext underlying the input ciphertext (for slots-encryption). We will not explicitly need homomorphic rotation in this work. Note that these algorithms preserve the ciphertext modulus q.

The homomorphic multiplication algorithm mult takes as input two ciphertexts for a common modulus q and a common scaling factor Δ. It outputs a ciphertext whose underlying plaintext is close to the coefficient-wise product of the plaintexts underlying the input ciphertexts (for slots-encryption). The modulus of the output ciphertext is $q' \approx q/\Delta$.

Note that for a given ring degree N, the ciphertext modulus q cannot exceed some value if one wants to maintain sufficient security. As multiplications decrease the ciphertext modulus, the number of multiplications that one can perform sequentially while only relying on the operations mentioned so far is bounded. For this reason, the CKKS literature always considers a chain of moduli $q_0 < q_1 < \ldots$ corresponding to multiplication levels. The integer q_0 is called the base modulus.

2.3 Bootstrapping

Bootstrapping allows one to regain modulus: it takes as input a ciphertext with a small modulus and outputs a ciphertext with higher modulus, whose underlying plaintext is close to the plaintext underlying the input ciphertext. Bootstrapping consists of the following four components.

- **Slots-to-Coefficients** (StC). Given a ciphertext that encrypts a complex vector $\mathbf{z} \in \mathbb{R}^{N/2}$ in its slots, we convert it to a ciphertext that encrypts $\mathbf{z}$ in its coefficients. This can be realized via homomorphic evaluation of DFT.
- **Modulus Raising** (ModRaise). Given a ciphertext ct at the base modulus q_0 encrypting a plaintext $m \in \mathcal{R}$, we regard it as a ciphertext encrypting $m + q_0 I \in \mathcal{R}$ without modulus. This increases the ciphertext modulus, while adding a small multiple of the base modulus.
- **Coefficients-to-Slots** (CtS). To prepare the removal of the $q_0 I$ term introduced in the coefficients by ModRaise, we convert the ciphertext to the slot-encoded format. This is realized with homomorphic evaluation of iDFT.
- **Homomorphic Modular Reduction** (EvalMod). We remove the $q_0 I$ term by homomorphically evaluating the "modulo-q_0" function (in a SIMD manner). Since modular reduction is discontinuous, one may set parameters in a way that ensures that there is a gap between m and q_0, so that it suffices to approximate the "modulo-q_0" function only on small intervals around integer multiples of q_0. This is achieved by a polynomial approximation to the $x \mapsto (q_0/2\pi) \cdot \sin(2\pi x/q_0)$ function.

In this work, we consider StC-*first bootstrapping*, which executes EvalMod $\circ$ CtS $\circ$ ModRaise $\circ$ StC. This requires to start the process at a modulus larger than q_0 so that StC is completed at the base modulus q_0.

Real bootstrapping is restricted to ciphertexts whose underlying plaintexts are in $\mathbb{R}^{N/2}$ (in slots). *Complex bootstrapping* works for plaintexts in $\mathbb{C}^{N/2}$. The main difference lies in the EvalMod function, whose correctness holds if its input ciphertext decrypts to a real vector. If the vector is complex, one extracts the real and imaginary parts using homomorphic conjugation just before EvalMod, runs EvalMod twice in parallel, and finally recombines the outputs into a single ciphertext that encodes the desired complex vector.

In [BCKS24], the EvalMod function was adapted to handle binary data ciphertexts (i.e., with $\mathbf{z} \in \{0,1\}^{N/2}$). At the bottom modulus, i.e., just before ModRaise, the ciphertexts encode $\mathbf{z}$ in their most significant bits, and a properly scaled version of the sine function is used to bootstrap (other variants are considered in [BCKS24], such as evaluating a binary gate and bootstrap at once).

3 Improving the Roots-of-Unity Toolbox

In this section, we consider the efficiency of some homomorphic computations involving roots-of-unity encodings.

We first revisit the conversion algorithm from [BGGJ20], from DM/CGGI ciphertexts to a CKKS ciphertext. We observe that the main step of this algorithm bootstraps a ciphertext for integers into a ciphertext for roots of unity. We provide a depth-consumption analysis, based on the observation that the data of interest is encrypted in the most significant bits. We then consider the task of evaluating look-up-tables using roots-of-unity encodings, and decrease the required depth compared to [CKKL24]. Finally, we introduce an extension of interpolation that also decreases the error of a roots-of-unity encoding, for a

multiplicative depth that is twice lower than the depth required to interpolate and clean, based on the tools from [CKKL24].

3.1 Revising Chimera's Conversion from DM/CGGI to CKKS

Below, we study an algorithm introduced in [BGGJ20, Section 4.1], in the context of converting multiple DM/CGGI ciphertexts to a CKKS ciphertext. This conversion algorithm handles three difficulties. The first one is of a packing nature: a DM/CGGI ciphertext contains a single small integer as plaintext, whereas a CKKS ciphertext can store up to N (when using coefficients-encoding, or slots-encoding with real and imaginary parts). In this subsection, we do not consider this aspect. The second difficulty is that a DM/CGGI ciphertext has a small modulus, typically with magnitude similar to the base modulus q_0 of CKKS. The third difficulty is of a scaling nature: in DM/CGGI, the plaintext lies in the most significant bits of the ciphertext (sometimes with some margin to allow for one homomorphic addition), whereas in a CKKS ciphertext, the most significant bits are typically not used. Such a plaintext scaling prevents the use of conventional CKKS bootstrapping, as it makes it very difficult to approximate the discontinuous "modulo-q_0" function (see Section 2.3). More concretely, a coefficients-encoded CKKS ciphertext $(a, b) \in \mathcal{R}_q^2$ with DM/CGGI plaintext scaling would be such that $a \cdot s + b \approx (q/t) \cdot z$, where s is the secret key and $z \in \mathcal{R}_t$ is the plaintext. Oppositely, a typical CKKS ciphertext would satisfy $a \cdot s + b \approx (\Delta/t) \cdot z$ with $\Delta \ll q$.

Putting aside the packing of multiple DM/CGGI ciphertexts, the algorithm from [BGGJ20, Section 4.1] can be revisited as a transformation from a low-level coefficients-encoding CKKS ciphertext for a plaintext in the most significant bits into a high-level roots-of-unity-encoding CKKS ciphertext. We revisit it as a CKKS bootstrapping algorithm from slots-encoded integers to slots-encoded roots of unity for the same data.

We assume that the input ciphertext decrypts to a vector (in slots) that corresponds to a vector $\mathbf{m} = \mathbf{z} + \boldsymbol{\varepsilon}$ in $\mathbb{C}^{N/2}$ where $\mathbf{z}$ is an integer vector with coefficients in $[0, t)$ and the error term $\boldsymbol{\varepsilon}$ satisfies $\|\boldsymbol{\varepsilon}\|_\infty \ll 1$. We choose the scaling factor Δ_0 at the base modulus q_0 as $\Delta_0 = q_0/t$, so that for ciphertexts modulo q_0, the message is coefficients-encoded in the most significant bits. IntRootBoot, described in Algorithm 1, proceeds as follows.

- It runs StC to put the message in the coefficients. The message is then placed in the most significant bits, by choice of Δ_0.
- It runs ModRaise to increase the ciphertext modulus. The message can then be described as $(q_0/t) \cdot m + q_0 \cdot I \in \mathcal{R}$ for some $I \in \mathcal{R}$. Note that it can be rewritten as $(q_0/t)(m + tI)$: the aim of the subsequent steps is to homomorphically reduce $m + tI$ modulo t to remove the tI term.
- It runs CtS to put the message in the slots. It uses homomorphic conjugation to compute the real part.
- It runs EvalExp, which is the homomorphic evaluation of the function $x \mapsto e^{2\pi i x/t}$. Note that after ModRaise, the message is interpreted as $m + tI$,

and the exponential function can be implemented using homomorphic conjugation and the trigonometric functions $x \mapsto \sin(2\pi x/t)$ and $\mapsto \cos(2\pi x/t)$ (using the identity $e^{ix} = \cos(x) + i\sin(x)$).

The above extends to a complex bootstrapping algorithm, i.e., for an input plaintext $\mathbf{m} = \mathbf{z} + \boldsymbol{\varepsilon}$ in $\mathbb{C}^{N/2}$ where both the real and imaginary parts of $\mathbf{z}$ are integer vectors with coefficients in $[0, t)$. This is achieved by appropriately adapting Steps 2 and 3 of Algorithm 1, as discussed in Section 2.3.

Algorithm 1: IntRootBoot

Setting: $\Delta_0 = q_0/t$.
Input : $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}}(\mathbf{z} + \boldsymbol{\varepsilon}) \in \mathcal{R}_q^2$ with $\mathbf{z} \in \{0, 1, \ldots, t-1\}^{N/2}$ and $\|\boldsymbol{\varepsilon}\|_\infty \ll 1$.
Output: $\mathsf{ct}_{\mathsf{out}} \in \mathcal{R}_Q^2$.

1 $\mathsf{ct}' \leftarrow \mathsf{CtS} \circ \mathsf{ModRaise} \circ \mathsf{StC}(\mathsf{ct})$;
2 $\mathsf{ct}'' \leftarrow (\mathsf{conj}(\mathsf{ct}') + \mathsf{ct}')/2$;
3 $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{EvalExp}(\mathsf{ct}'')$;
4 **return** $\mathsf{ct}_{\mathsf{out}}$.

Let $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}}(\mathbf{z} + \boldsymbol{\varepsilon}) \in \mathcal{R}_q^2$ be an input ciphertext encoding an integer vector $\mathbf{z} \in \{0, 1, \ldots, t-1\}^{N/2}$ with an error $\boldsymbol{\varepsilon}$ satisfying $\|\boldsymbol{\varepsilon}\|_\infty \ll 1$. Assume that homomorphic operations StC, ModRaise, StC and conj and EvalExp give sufficiently high precision. Then the output $\mathsf{ct}_{\mathsf{out}}$ of Algorithm 1 encrypts the vector $(e^{2\pi i(z_j+\varepsilon_j)/t})_{0 \le j < N/2}$, up to a tiny error. The latter is close to $e^{2\pi i z_j/t}$. Indeed, the difference can be bounded from above as follows, for all j:

$$\begin{aligned}
|e^{2\pi i(z_j+\varepsilon_j)/t} - e^{2\pi i z_j/t}| &= |e^{2\pi i z_j/t}| \cdot |e^{2\pi i \varepsilon_j/t} - 1| \\
&= |e^{2\pi i \varepsilon_j/t} - 1| \\
&\le |\sin(2\pi\varepsilon_j)| + |\cos(2\pi\varepsilon_j) - 1| \\
&= |\sin(2\pi\varepsilon_j)| + |2\sin^2(\pi\varepsilon_j)| \\
&\le 2\pi\|\boldsymbol{\varepsilon}\|_\infty + 2\pi^2\|\boldsymbol{\varepsilon}\|_\infty^2 \ .
\end{aligned}$$

Apart from EvalExp, all operations are as in CKKS bootstrapping. EvalExp can be performed using the formula $e^{ix} = \cos(x) + i\sin(x)$. Homomorphic evaluations of sin and cos have been extensively explored throughout the CKKS literature (see, e.g., [CHK+18,LLL+21]), as it is a key ingredient of CKKS bootstrapping algorithms. An approach is to perform EvalExp by homomorphically evaluating the sine function twice (once for $\sin(2\pi x)$ and once for $\cos(2\pi x) = \sin(\pi/4 - 2\pi x)$). The cost is then roughly twice that of EvalMod. As EvalMod and CtS have similar costs (the other bootstrapping steps being less costly), the total cost of IntRootBoot is a little larger than the cost of the conventional CKKS bootstrapping.

We now argue that IntRootBoot consumes relatively little modulus. Since conventional CKKS bootstrapping and IntRootBoot both evaluate StC, CtS and

EvalMod/EvalExp, the multiplicative depth consumption is the same. However, as there is no gap between message and modulus at the base level (corresponding to modulus q_0), we may use relatively smaller scaling factors during CtS and EvalMod, leading to a reduction of modulus consumption. For concreteness, let us assume that we are interested in integers in $[0, t)$, with an accuracy of $\gamma_{acc} = \log \|\varepsilon\|_\infty$ bits, and an FHE computing noise of γ_{noise}. The FHE noise is typically slightly above $(\log N + \log h)/2$, where h is the Hamming weight of the secret key (to fix the ideas, one may consider that $\gamma_{noise} = 12$). For each level in CtS and EvalExp, the scaling factor must have $\approx \log(2I_{\max}) + \log t + \gamma_{acc} + \gamma_{noise}$ bits, to represent the I term with coefficients in $[-I_{\max}, I_{\max}]$, the integer vector $\mathbf{z}$ under scope, the accuracy bits and the FHE computing noise. By using the sparse secret encapsulation technique from [BTPH22], the integer I belongs to $[-15, 15]$ with probability extremely close to 1, leading to $\log(2I_{\max}) = 5$. In the StC levels, there is no need for these first 5 bits, as the I term vanishes due to the use of appropriate roots of unity. This gives the following rough approximation to modulus consumption:

$$\begin{aligned}\mathsf{ModCons}_{\mathsf{IntRootBoot}} \approx (\ell_{\mathsf{CtS}} + \ell_{\mathsf{EvalExp}}) \cdot (5 + \log t + \gamma_{acc} + \gamma_{noise}) \\ + \ell_{\mathsf{StC}} \cdot (\log t + \gamma_{acc} + \gamma_{noise}) \quad ,\end{aligned}$$

where $\ell_{\mathsf{CtS}}, \ell_{\mathsf{EvalExp}}$ and ℓ_{StC} respectively refer to the multiplicative depths of CtS, EvalExp and StC.

By using conventional bootstrapping, one would need to consider a gap to encode the integer to be bootstrapped, to make it small compared to the bottom modulus, to enable a polynomial approximation to the "modulo-q_0" function. In practice, one often chooses a gap of $\gamma_{gap} \approx 10$ bits. This gap is added to the modulus consumption of all levels of CtS and EvalExp. This implies that IntRootBoot consumes $\approx (\ell_{\mathsf{CtS}} + \ell_{\mathsf{EvalExp}}) \cdot \gamma_{gap}$ fewer bits of modulus than conventional CKKS bootstrapping. For bootstrapping techniques that are currently used, this most often amounts to more than 100 bits.

3.2 Interpolation for Roots of Unity

We now consider the task of homomorphically evaluating a look-up table, from the $e^{2\pi ij/t}$'s for some integer $t \geq 2$ and $j \in \{0, 1, \ldots, t-1\}$ to arbitrary complex values $(y_j)_{0 \leq j < t}$. Concretely, we aim at finding a function f such that $f(e^{2\pi ij/t}) = y_j$ for all $0 \leq j < t$, which can be homomorphically evaluated with good efficiency and low modulus consumption. It was observed in [CKKL24] that complex t-th roots of unity provide good numerical stability when it comes to polynomial interpolation. We argue below that complex conjugation can help performing look-up table evaluations on such points.

Suppose that $f : x \mapsto f_0 + f_1 x + \ldots + f_{t-1} x^{t-1}$ is the Lagrange interpolation from the $e^{2\pi ij/t}$'s to the y_j's. In full generality, evaluating f requires depth $\log t$.

Now, note that we may as well evaluate

$$g : x \mapsto \left(f_0 + f_1 x + \ldots + f_{\lfloor t/2 \rfloor} x^{\lfloor t/2 \rfloor}\right) + \left(f_{t-1}\overline{x} + f_{t-2}\overline{x}^2 + \ldots f_{\lfloor t/2 \rfloor + 1}\overline{x}^{t-(\lfloor t/2 \rfloor + 1)}\right) \ .$$

Indeed, on the $e^{2\pi ij/t}$'s, the values x^i and $\overline{x}^{t-i}$ coincide. Note that g requires a homomorphic conjugation. As homomorphic conjugation does not consume depth and g has degree twice less than f, it can be evaluated with one less multiplicative depth.

3.3 Combined Interpolation and Cleaning for Roots of Unity

We now describe another interpolation strategy, which simultaneously interpolates and increases the accuracy of the discrete data points. A first approach would be to use Hermite interpolation, i.e., extending Lagrange interpolation with the condition that the derivative of the polynomial cancels on the interpolation points. The cancelling derivatives imply a decrease of the noise, similarly to the h_1 function from Section 2.1. As there are twice more conditions, this polynomial has degree $< 2t$. Note that the complex conjugation approach of the previous subsection does not apply. It would result in a bivariate polynomial in x and $\bar{x}$ which is not differentiable. It preserves the evaluations of the initial polynomial, but there is no a priori reason for the noise-cleaning functionality to be preserved. For example, the polynomial $f(x) = 3x/2 + x^3/2$ cleans for inputs in $\{-1, 1\}$. However, the function $g(x) = 3x/2 + \overline{x}/2$ is equal to f on $\{-1, 1\}$ but without any cleaning functionality since $g(1 + e) = 1 + 3e/2 - \overline{e}/2$ has non-vanishing linear terms in e and $\overline{e}$.

We now explain how to exploit complex conjugation to obtain a combined interpolation and noise-cleaning functionality. Let f be the Lagrange interpolation polynomial, from the roots of unity to the desired y_j's. We consider the following function, which may be viewed as a bivariate polynomial in x and $\overline{x}$:

$$\begin{aligned} h : x \mapsto & f_0 + \sum_{k=1}^{\lfloor t/2 \rfloor} \frac{f_k}{t}\left(k\overline{x}^{t-k} + (t-k)(k+1)x^k - k(t-k)x^{k+1}\overline{x}\right) \\ & + \sum_{k=\lfloor t/2 \rfloor + 1}^{t-1} \frac{f_k}{t}\left((t-k)x^k + k(t-k+1)\overline{x}^{t-k} - k(t-k)\overline{x}^{t-k+1}x\right) \end{aligned}$$

Observe that each f_i is multiplied by a trinomial in x and $\overline{x}$ such that each monomial $x^a\overline{x}^b$ satisfies $a - b = k \bmod t$. The trinomials are chosen so that the evaluation of any of these in the t-th roots of unity is equal to 1, so that the evaluation of h is exactly the same as that of f. Further, the trinomials are such that when evaluated in $x + \varepsilon$ and $\overline{x} + \overline{\varepsilon}$, the partial derivatives with respect to ε and $\overline{\varepsilon}$ cancel in the t-th roots of unity. This property provides the cleaning functionality. We stress that there is flexibility in the choice of the monomials

appearing in the trinomials, and that we opted to have a pattern, as well as some symmetry between the first half and the second half. The total degree of h is $\max(t-1, 4)$, and h can be evaluated with multiplicative depth $\log t$ (when $t \geq 4$), i.e., one less than the polynomial obtained with Hermite interpolation.

Lemma 2. *There exists a constant $C > 0$ such that the following holds. Let $t \geq 2$ and $y_0, \ldots, y_{t-1} \in \mathbb{C}$. Let f be the univariate polynomial of degree $< t$ such that $f(e^{2\pi ij/t}) = y_j$ for all $0 \leq j < t$, and h be as above. Then, for all $0 \leq j < t$ and $\varepsilon \in \mathbb{C}$ with $|\varepsilon| \leq 1/(t-1)$, we have:*

$$\left| h(e^{2\pi ij/t} + \varepsilon) - y_j \right| \leq C \cdot t^3 \cdot \max_j |f_j| \cdot |\varepsilon|^2 \ .$$

Proof. Let $\zeta = e^{2\pi ij/t}$ for some arbitrary $0 \leq j < t$. Replacing x by $\zeta + \varepsilon$ in the definition of h, we see that it can be expressed as a bivariate polynomial in ε and $\overline{\varepsilon}$. By using the relation $\overline{\zeta} = \zeta^{-1}$ and using the definition of f, we obtain that the constant term of that bivariate polynomial is

$$h(\zeta) = f_0 + f_1 \cdot \zeta + \ldots + f_{t-1}\zeta^{t-1} = y_j \ ,$$

where the second equality is by definition of f.

It may then be checked that the terms linear in ε and $\overline{\varepsilon}$ sum to 0. Therefore, in order to bound $|h(\zeta+\varepsilon) - y_j|$, it suffices to consider the terms of total degree 2 or more in ε and $\overline{\varepsilon}$. For this purpose, we will use the following inequality, which holds for all integer $n \geq 2$, all $x \in \mathbb{C}$ with $|x| \leq 1$ and all $u \in \mathbb{C}$ with $|u| \leq 1/n$:

$$\left| (x+u)^n - x^n - nx^{n-1}u \right| \leq n^2 |u|^2 \ .$$

By using the facts that $|\zeta| = 1$ and $|\varepsilon| \leq 1/(t-1)$, the triangle inequality and the above inequality, we have:

$$|h(\zeta + \varepsilon) - y_j| \leq C \cdot t^3 \cdot \max_j |f_j| \cdot |\varepsilon|^2 \ ,$$

for some (absolute) constant C. □

An interesting particular case is the interpolation for the identity function. This provides a cleaning functionality. Taking $f(x) = x$ in the above definition of h, we obtain the function:

$$x \mapsto \frac{1}{t}\left(\overline{x}^{t-1} + 2(t-1)x - (t-1)x^2\overline{x}\right) \ .$$

It has degree $\max(3, t-1)$. It may be compared to the cleaning polynomial $x \mapsto ((t+1)x - x^{t+1})/t$ considered in [CKKL24], of degree $t+1$. For t's chosen as powers of two, this provides a saving of one multiplicative depth.

3.4 Polynomial Multi-Evaluation

We now consider the task of evaluating the bivariate polynomials of Sections 3.2 and 3.3, by viewing it as a variant of homomorphically evaluating several polynomials on the same input.

Consider first the case of a single polynomial $P = P_0 + P_1 x + \ldots + P_{d-1} x^{d-1}$. A naive version of the Paterson-Stockmayer algorithm [PS73] proceeds as follows:

1. (Initialization) Compute $1, x, \ldots, x^{\sqrt{d}-1}$ and $x^{\sqrt{d}}, x^{2\sqrt{d}}, x^{2^2\sqrt{d}}, \ldots$;
2. (Baby steps) Compute

$$\begin{aligned} \pi_0 &= P_0 + P_1 x + \ldots + P_{\sqrt{d}-1} x^{\sqrt{d}-1} \ , \\ &\vdots \\ \pi_{\sqrt{d}-1} &= P_{d-\sqrt{d}} + P_{d-\sqrt{d}+1} x + \ldots + P_{d-1} x^{\sqrt{d}-1} \ ; \end{aligned}$$

3. (Giant steps) Compute $\pi_0 + \ldots + x^{d-\sqrt{d}} \pi_{\sqrt{d}-1}$ with a binary recursion.

Homomorphically, this amounts to $\approx 2\sqrt{d}$ ciphertext-ciphertext multiplications, half of them in the initialization and the other half in the giants steps. As ciphertext-ciphertext multiplications are significantly more costly than plaintext-ciphertext multiplications, this dominates the cost. Note further that the multiplicative depth is $\log d$.

Now, consider a scenario in which we would like to homomorphically evaluate k polynomials $P^{(0)}, \ldots, P^{(k-1)}$ on the same ciphertext. By applying the above agorithm k times, one obtains a cost dominated by $\approx 2k\sqrt{d}$ ciphertext-ciphertext multiplications. Now, observe that the initialization can be shared across the polynomial evaluations, the number of ciphertext-ciphertext multiplications can be decreased to $\approx (k+1)\sqrt{d}$. This can be decreased further by modifying the balance between the baby steps and giant steps, as follows.

1. (Initialization) Compute $1, x, \ldots, x^{\sqrt{kd}-1}$ and $x^{\sqrt{kd}}, x^{2\sqrt{kd}}, x^{2^2\sqrt{kd}}, \ldots$;
2. (Baby steps) For all $0 \leq i \leq \sqrt{d/k}$ and $0 \leq j \leq k$, compute

$$\pi_i^{(j)} = P^{(j)}_{i\sqrt{kd}} + P^{(j)}_{1+i\sqrt{kd}} x + \ldots + P^{(j)}_{\sqrt{d}-1+i\sqrt{kd}} x^{\sqrt{kd}-1} \ ;$$

3. (Giant steps) For all $0 \leq j \leq k$, compute $\pi_0^{(j)} + x^{\sqrt{kd}} \pi_1^{(j)} + \ldots + x^{d-\sqrt{kd}} \pi^{(j)}_{\sqrt{kd}-1}$ using $x^{\sqrt{kd}}, x^{2\sqrt{kd}}, x^{3\sqrt{kd}}, \ldots$ and a binary recursion.

The number of ciphertext-ciphertext multiplications then decreases to $\approx 2\sqrt{kd}$. At the same time, the multiplicative depth is preserved.

In Section 3.2, the function to be evaluated is the sum of a polynomial in x and a polynomial in $\overline{x}$. This may be handled similarly as above for $k = 2$, by computing the baby step for x and applying homomorphic conjugation on its output. Note that homomorphic conjugation adds a small cost that is independent of the degree d.

The function h in Section 3.3 can be handled similarly, with four polynomial evaluations. Indeed, it can be expressed as:

$$h(x) = f_0 + (P_f(x) + P_{f^r}(\bar{x})) - \bar{x}x\,(Q_f(x) + Q_{f^r}(\bar{x})) \ ,$$

where the polynomial f^r is the reversal of f, i.e., with $f^r_k = f_{t-k}$ for $k \in \{1, \ldots, t-1\}$, and:

$$P_f(x) = \sum_{k=1}^{t-1} \alpha_{f,k} x^k \ \text{ with } \ \alpha_{f,k} = \begin{cases} \frac{f_k}{t}(t-k)(k+1) & \text{if } k \leq t/2 \\ \frac{f_k}{t}(t-k) & \text{otherwise} \end{cases} \ ,$$

$$Q_f(x) = \sum_{k=1}^{\lfloor t/2 \rfloor} -\frac{f_k}{t} k(t-k) x^k \ .$$

4 Bootstrapping Small Integers

We now present our bootstrapping algorithm for small integers, as well as its extension to multi-function functional bootstrapping and its application to batch-bootstrapping multiple DM/CGGI ciphertexts for small integers.

4.1 SI-BTS

Assume that the plaintext underlying the input ciphertext is a vector $\mathbf{m}$ of small integers, between 0 and $t-1$ for some $t \geq 2$, and one wants to obtain a ciphertext whose plaintext is also a vector $\mathbf{y}$ of small integers, so that $y_j = f(m_j)$ for all $0 \leq j < N/2$. Here f is an arbitrary function from integers in $[0, t)$ to integers in $[0, t)$. (We could consider different sets for inputs and outputs, but keep them identical for the sake of simplicity.)

For this purpose, SI-BTS (given in Algorithm 2) first uses IntRootBoot, and then interpolate from the t-th root of unity $\exp(2i\pi m/t)$ to the integer $f(m)$ by using the combined interpolation and cleaning from Section 3.3. If cleaning is not necessary (for example, if it occurred soon earlier in the computations or is to be performed soon after), then one may optionally rely on the interpolation algorithm from Section 3.2. We however recall that it is interesting to clean together with a polynomial interpolation on the roots of unity, as it consumes only one additional multiplicative level compared to only interpolating (compared to $\approx \log t$ levels for a separate cleaning).

Converting from roots-of-unity embedding to integer embedding is important to be able to run SI-BTS again. However, in some cases, it may be interesting to postpone this conversion rather than performing it in bootstrapping, and keep the roots-of-unity embedding for a while. For example, for evaluating table look-ups, roots-of-unity embedding has been showed quite advantageous (see [CKKL24]). In this case, one can replace the interpolation of Step 2 by the one that sends $\exp(2i\pi x/t) \mapsto \exp(2i\pi f(x)/t)$ for all $0 \leq x < t$.

Correctness follows by inspection. We now adapt the modulus consumption of the end of Section 3.1 to the SI-BTS algorithm. In Section 3.1, we already

Algorithm 2: Small Integer (Functional) Bootstrapping (SI-BTS)

Input : A CKKS ciphertext decrypting to $\approx \mathbf{m} \in \mathbb{C}^{N/2}$ in the slots, where $m_j \in \mathbb{Z} \cap [0, t)$ for all $0 \leq j < N/2$; a function $f : \mathbb{Z}_t \to \mathbb{Z}_t$.
Output: A CKKS ciphertext whose modulus is no smaller, and decrypting to $\approx (f(m_j))_j \in \mathbb{C}^{N/2}$ in the slots.
Keys : IntRootBoot bootstrapping, conjugation and relinearization keys.

1 Run IntRootBoot on ct;
2 Homomorphically evaluate, with cleaning, the function that maps $\exp(2i\pi x/t)$ to $f(x)$ for all $0 \leq x < t$; for this purpose, use Section 3.3; let ct be the output ciphertext;
3 **return** ct.

estimated the modulus consumption of Step 1. As seen in Section 3.3, we may interpolate and clean at Step 2, with $\log t$ multiplicative levels. At that stage, we do not need to represent the integer I any more, so that a level corresponds to the same amount of modulus as in StC.

$$\begin{aligned} \mathsf{ModCons}_{\mathsf{SI\text{-}BTS}} \approx (\ell_{\mathsf{CtS}} + \ell_{\mathsf{EvalExp}}) \cdot (5 + \log t + \gamma_{acc} + \gamma_{noise}) \\ +(\ell_{\mathsf{StC}} + \log t) \cdot (\log t + \gamma_{acc} + \gamma_{noise}) \ , \end{aligned} \tag{1}$$

where ℓ_{CtS}, ℓ_{EvalExp} and ℓ_{StC} respectively refer to the multiplicative depths of CtS, EvalExp and StC. The quantity γ_{noise} corresponds to the bit-size of the noise induced by homomorphic operations and γ_{acc} corresponds to the accuracy of the representations of the $(\log t)$-bit integers. We insist that this estimate is rough, and we refer the reader to Section 6 for concrete experimental data. We note that γ_{acc} is not constant throughout the computation: it first decreases because of homomorphic computations, and it is then replenished at Step 2. We do not consider this variation in (1). Similarly, the quantity γ_{noise} varies depending on the type of homomorphic operations performed. It can be seen in (1) that when t is small, the modulus consumption grows very slowly, as the terms linear in $\log t$ are 'somewhat hidden' by $\gamma_{acc} + \gamma_{noise}$, and, to a lesser extent, by $\ell_{\mathsf{CtS}} + \ell_{\mathsf{StC}} + \ell_{\mathsf{EvalExp}}$. However, as t increases, the growth rate eventually becomes quadratic in $\log t$.

In terms of cost, the situation is similar. For small t, the cost will be dominated by Step 1, but when t increases, the cost of Step 2 will eventually become dominant.

4.2 Multi-Output SI-BTS

The SI-BTS algorithm can be extended to evaluate several functions f_i for a given input m. This may be viewed as a CKKS (and hence SIMD) analogue to the DM/CGGI multi-output bootstrap algorithm from [CIM19].

Algorithm 2 is then modified as follows. Step 1 is run only once, while Step 2 can benefit from the multi-evaluation algorithm of Section 4.2. Overall, if K

is the number of functions being evaluated in parallel, the cost of Step 1 is independent of K, while the cost of Step 2 essentially grows with $\sqrt{K}$ (the ciphertext-ciphertext multiplications being the most expensive component of polynomial evaluation). For a small K and a small t, the cost of Step 2 is limited compared to the cost of Step 1, so that several functional bootstraps can be performed for essentially the same cost as a single one. When t and K are larger, Step 2 dominates and the cost increase is more visible.

4.3 Batch Functional Bootstrapping of DM/CGGI Ciphertexts

SI-BTS is particularly useful when one wishes to perform multiple DM/CGGI functional bootstraps in parallel. This gives an extension of the batch DM/CGGI bootstrapping from [BCKS24], which was restricted to evaluating a binary gate.

Recall that a CGGI/DM ciphertext can be viewed as an LWE version of our coefficient-integer-encoded ciphertext, where the plaintext integer lies in the most significant bits of the ciphertext. More concretely, a ciphertext $\mathsf{ct} \in \mathbb{Z}_q^n$ for some integers n and q decrypts to an integer $m \in [0, t)$ under a key $\mathsf{sk} \in \{-1, 0, 1\}^n$ if:

$$\langle \mathsf{ct}, \mathsf{sk} \rangle = \frac{q}{t} m + e \bmod q \ ,$$

where $|e| \ll q/t$. For efficiency purposes, the modulus q is typically very small (e.g., it can have 12 bits).

Now, assume we are given $\leq N$ such ciphertexts $(\mathsf{ct}_j)_{0 \leq j < N}$, decrypting to integers $(m_j)_{0 \leq j < N}$ under a common key sk. These ciphertexts can be packed into a single coefficients-encoded CKKS ciphertext, by relying on a ring-packing procedure (see [CGGI17,BCK+23], among others). This provides a ciphertext $(a, b) \in R_{q_0}^2$ for the base CKKS modulus q_0 and for a key $s \in R$ such that

$$a \cdot s + b \approx \frac{q'}{t} \left(m_0 + m_1 \cdot X + \ldots + m_{N-1} X^{N-1} \right) \ .$$

Note that the base CKKS modulus q_0 is typically larger than q. The change of modulus from q to q_0 is implemented by scaling and rounding. Ring packing requires a dedicated evaluation key (some form of CKKS encryption of sk under s).

One then uses IntRootBoot (without StC) to bootstrap this CKKS ciphertext and evaluate an arbitrary function by interpolation. Complex bootstrapping may be used if there are $> N/2$ input ciphertexts ct_j. Finally, we run StC to put the message back in the coefficients, and by properly rearranging the coefficients of the obtained RLWE ciphertext, we obtain the desired LWE-format DM/CGGI ciphertexts.

5 Batch Bits Bootstrapping

In the previous section, we have seen how to bootstrap ciphertexts whose underlying plaintexts encode small integers, for a cost (mostly driven by bootstrapping

modulus consumption) that is not significantly higher than that of bootstrapping bits. It is hence tempting to use such an approach to batch-bootstrap bits, by packing them into integers, to increase the throughput for binary circuits.

5.1 BB-BTS

Let us assume we aim at simultaneously bootstrapping $k \geq 1$ CKKS ciphertexts whose underlying plaintexts correspond to bits, encrypted into slots. The goal is to bootstrap these ciphertexts together, for a cost that is significantly lower than bootstrapping them individually using the algorithm from [BCKS24].

Our approach is as follows. We first pack the data into a single ciphertext. More concretely, we create a single ciphertext such that for any $j \leq N/2$, the j-th slot contains a k-bit integer obtained by concatenating the bits in the j-th slots of the input ciphertexts. Then we apply $\mathsf{IntRootBoot}$ with 2^k-th roots of unity, to obtain a roots-of-unity slots-encoded ciphertext. The next step is to extract the individual bits from the roots-of-unity ciphertexts by running k interpolations (one for each bit). Finally, we decrease the noise of the resulting slots-encoded ciphertexts for bits, by using the h_1 cleaning function (see Section 2). This procedure is summarized in Algorithm 3.

Algorithm 3: Batch Bits Bootstrapping ($\mathsf{BB\text{-}BTS}$)

Input : $k \geq 1$ slots-encoded ciphertexts $\mathsf{ct}_0, \ldots, \mathsf{ct}_{k-1}$ for vectors in $\{0,1\}^{N/2}$.

Output: $k \geq 1$ slots-encoded ciphertexts $\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{k-1}$ for the same vectors of bits, at a higher modulus.

Keys : $\mathsf{IntRootBoot}$ bootstrapping, conjugation and relinearization keys.

1 Homomorphically evaluate $m_0, \ldots, m_j \mapsto \sum_j 2^j m_j$ on the input ciphertexts; let ct be the output ciphertext;

2 $\mathsf{ct} \leftarrow \mathsf{IntRootBoot}(\mathsf{ct})$;

3 For all $0 \leq j < k$, set ct'_j as the ciphertext obtained by homomorphically interpolating from $\exp(2\pi i(\sum_{0 \leq \ell < k} b_\ell 2^\ell)/2^k)$ to b_j;

4 For all $0 \leq j < k$, set ct'_j as the ciphertext obtained by homomorphically evaluating h_1 on ct'_j;

5 **return** $\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{k-1}$.

Correctness follows from inspection. We now analyze modulus consumption, by adapting the end of Section 3.1. Step 2 has been studied in Section 3.1. As seen in Section 3.2, we may interpolate at Step 3 with $k-1$ multiplicative levels. At that stage, we do not need to represent the integer I any more, but we still need to represent k-bit data points. Finally, evaluating the h_1 polynomial requires two multiplicative levels, and the data of interest has a single bit at that

stage. Overall, the modulus consumption is as follows:

$$\begin{aligned}\mathsf{ModCons}_{\mathsf{BB\text{-}BTS}} \approx\ & (\ell_{\mathsf{CtS}} + \ell_{\mathsf{EvalExp}}) \cdot (5 + k + \gamma_{acc} + \gamma_{noise}) \\ & + (k-1) \cdot (k + \gamma_{acc} + \gamma_{noise}) \\ & + (\ell_{\mathsf{StC}} + 2) \cdot (1 + \gamma_{acc} + \gamma_{noise}) \ \ ,\end{aligned} \tag{2}$$

where the variables $\ell_{\mathsf{CtS}}, \ell_{\mathsf{EvalExp}}, \ell_{\mathsf{StC}}, \gamma_{acc}$ and γ_{noise} are as before. It may seem that cleaning the ciphertexts may not be necessary, depending on how noisy they currently are. Indeed, if their noise is limited, then one may first evaluate some binary gates and postpone cleaning. However, cleaning at the end of bootstrapping allows us to lower the bootstrapping modulus consumption, and we argue that it should hence be viewed as a component of bootstrapping. Concretely, the quantity γ_{acc} is smaller for Steps 2 and 3 than it is at Step 4. When k is small, its impact is limited, because of the terms $\gamma_{acc} + \gamma_{noise}$, for the precision, and $\ell_{\mathsf{CtS}} + \ell_{\mathsf{EvalExp}} + \ell_{\mathsf{StC}} + 2$, for the multiplicative depth. However, when k increases, the modulus consumption eventually grows quadratically in k.

For the cost, the situation is similar. For small k, one expects Step 2 to dominate the cost. When k increases, the cost of Step 2 remains almost constant, but those of Steps 3 and 4 grow. The highest throughput is achieved when the cost and modulus consumption of these steps is correctly balanced with the cost and modulus consumption of Step 2.

We note that using complex bootstrapping allows to improve throughput further, as CtS and StC can then handle twice more data for the same cost. However, EvalExp and Steps 2 and 3 of BB-BTS are then run twice in parallel.

5.2 Extracting bits

It could be tempting to view Step 3 as a multi-evaluation and use the algorithm described in Section 4.2. One could even avoid Step 4 by cleaning the noise in Step 3, by using the approach given in Section 3.3, and save one multiplicative depth. However, it seems preferable to exploit the fact that the interpolation polynomials for extracting bits are not generic at all. For example, the least significant bit b_0 can be obtained from $\exp(2i\pi(\sum_{0\leq \ell<k} b_\ell 2^\ell)/2^k)$ by raising it to the 2^{k-1}-th power to obtain $(-1)^{b_0} = -2b_0 + 1$ and then correct the result to b_0. As showed in the following lemma, the interpolation polynomials for the subsequent bits are also very sparse.

Lemma 3. *Let $k \geq 1$ and $0 \leq j < k$. Define $P_{k,\ell} \in \mathbb{C}[x]$ as the minimal degree polynomial that maps $\exp(2i\pi(\sum_{0\leq \ell<k} b_\ell 2^\ell)/2^k)$ to b_j, for all $b_0, \ldots, b_{k-1} \in \{0,1\}$. Then $P_{k,\ell}$ has at most $1 + 2^\ell$ non-zero coefficients, for monomials of degrees 0 and odd multiples of 2^{k-j-1} that are $< 2^k$.*

Proof. We prove the result by induction on k. It may be checked that it holds for $k = 1$, and we now assume that $k \geq 2$. We now consider two cases, depending on the value of j. Assume first that $j < k-1$. Note that for any bits $b_0, \ldots, b_{k-1}$, raising the 2^k-th root of unity $\exp(2i\pi(\sum_{0\leq \ell<k} b_\ell 2^\ell)/2^k)$ to the power 2^{k-j-1}

gives the 2^{j+1}-th root of unity $\exp(2i\pi(\sum_{0\leq\ell\leq j} b_\ell 2^\ell)/2^{j+1})$. By unicity of interpolating polynomials of small degree, we then obtain that

$$P_{k,j}(x) = P_{j+1,j}(x^{2^{k-j-1}}) \ .$$

The induction hypothesis gives the result. We now consider the remaining case, i.e., $j = k-1$. Using the observation that

$$\exp\left(\frac{2i\pi}{2^k}\left(2^{k-1} + \sum_{0\leq\ell<k-1} b_\ell 2^\ell\right)\right) = -\exp\left(\frac{2i\pi}{2^k}\left(0 + \sum_{0\leq\ell<k-1} b_\ell 2^\ell\right)\right) \ ,$$

we observe that for any θ in the set of interpolating points, we have that $-\theta$ belongs to the set of interpolating points and one is mapped to $0 = -1/2 + 1/2$ whereas the other one is mapped to $1 = 1/2 + 1/2$. This implies that the shifted interpolation polynomial $P_{k,k-1} - 1/2$ is odd. In particular, its non-zero coefficients can only be for odd powers of x. □

As an illustration, we give the list of $P_{4,\ell}$'s below.

$$\begin{aligned}
P_{4,0} &= \frac{1}{2} - \frac{1}{2}x^8 \ , \\
P_{4,1} &= \frac{1}{2} + \alpha_4 x^4 + \alpha_{12} x^{12} \ , \\
P_{4,2} &= \frac{1}{2} + \alpha_2 x^2 + \alpha_6 x^6 + \alpha_{10} x^{10} + \alpha_{14} x^{14} \ , \\
P_{4,3} &= \frac{1}{2} + \alpha_1 x + \alpha_3 x^3 + \alpha_5 x^5 + \alpha_7 x^7 + \alpha_9 x^9 + \alpha_{11} x^{11} + \alpha_{13} x^{13} + \alpha_{15} x^{15} \ ,
\end{aligned}$$

where:

$$\begin{bmatrix}\alpha_4\\ \alpha_{12}\end{bmatrix} = \begin{bmatrix}-0.25+0.25i\\ -0.25-0.25i\end{bmatrix} \ , \ \begin{bmatrix}\alpha_2\\ \alpha_6\\ \alpha_{10}\\ \alpha_{14}\end{bmatrix} \approx \begin{bmatrix}-0.125+0.3018i\\ -0.125+0.0518i\\ -0.125-0.0518i\\ -0.125-0.3018i\end{bmatrix} \ , \ \begin{bmatrix}\alpha_1\\ \alpha_3\\ \alpha_5\\ \alpha_7\\ \alpha_9\\ \alpha_{11}\\ \alpha_{13}\\ \alpha_{15}\end{bmatrix} \approx \begin{bmatrix}-0.0625+0.3142i\\ -0.0625+0.0935i\\ -0.0625+0.0418i\\ -0.0625+0.0124i\\ -0.0625-0.0124i\\ -0.0625-0.0418i\\ -0.0625-0.0935i\\ -0.0625-0.3142i\end{bmatrix} \ .$$

We now describe a multi-evaluation algorithm specifically designed for evaluating the $P_{k,\ell}$'s for $0 \leq \ell < k$. The shape of the polynomials to be evaluated makes it suitable for an adapted version of the Paterson-Stockmeyer algorithm [PS73]. It is possible to use the algorithm described in Section 4.2 to evaluate these multiple polynomials, but such an approach wastes the potential speed-ups stemming from the **sparsity** of the coefficients, as well as the fact that $\alpha_i X^i = \overline{\alpha_{16-i} X^{16-i}}$ for every i. We instead write the polynomial $P_{k,\ell}$ as

$$P_{k,\ell}(X) = \frac{1}{2} + X^{2^{k-\ell-1}} Q_{k,\ell}(X^{2^{k-\ell}}) + \overline{X^{2^{k-\ell-1}} Q_{k,\ell}(X^{2^{k-\ell}})} \ ,$$

with $\deg(Q_{k,\ell}) \leq 2^{\ell-1}$. We then sequentially use the Paterson-Stockmeyer algorithm evaluating a polynomial of degree d in $2\sqrt{d}$ non-scalar multiplications, for each $Q_{k,\ell}$. The total number of non-scalar multiplications for evaluating all polynomials follows a geometric sum of ratio $1/\sqrt{2}$, since $\lfloor \deg(Q_{k,\ell})/2 \rfloor = \deg(Q_{k,\ell-1})$. The resulting number of ciphertext-ciphertext multiplications is $\simeq 3.4 \cdot 2\sqrt{2^{k-2}}$.

To further optimize the algorithm for extracting bits, one can recycle the setup basis to evaluate all polynomials $Q_{k,\ell}$'s. Assume that $Q_{k,\ell}$ has degree 2^{2u-1}. The baby-step phase of the Paterson-Stockmeyer algorithm computes $1, x, x^2, \ldots, x^{u-1}, x^u, x^{2u}, x^{4u}, \ldots$. Since $Q_{k,\ell-1}(X^2) = Q_{k,\ell}(X) \bmod (X^{2^k} - 1)$ for $1 \leq \ell < k$ with $\deg(Q_{k,\ell-1}) = \lfloor \deg(Q_{k,\ell})/2 \rfloor$, half of the baby-step basis (the odd-indexed elements) becomes useless. The other half is still sufficient to run the Paterson-Stockmeyer algorithm. The baby-step phase is thus recycled across the polynomial evaluations. However, the baby-step basis may be unbalanced compared to the giant-step basis after too many recyclings. This strategy alone is suboptimal to minimize the number of nonscalar multiplications. To handle this issue, we extend the baby-steps basis, which results in twice less recursive calls for polynomial evaluation. The halved baby-step basis $1, x^2, \ldots, x^{u-2}$ becomes extended to $1, x^2, \ldots, x^{u-2}, x^u, \ldots, x^{2u-2}$ and the giant-step basis starts now at x^{2u} as described in Algorithm 4.

Algorithm 4: BitExtract

Setting: Compute $u \leftarrow 2^{\lfloor \log 2(\sqrt{t}) \rfloor}$, $bs = \{1, x^2, x^4, x^6, \ldots, x^{u-2}\}$ and $gs = \{x^u, x^{2u}, x^{4u}, x^{8u}, \ldots, x^{2^{v-1}u}\}$.
Input : $x \leftarrow \mathsf{Enc}_{\mathsf{sk}}(\mathbf{z} + \boldsymbol{\varepsilon}) \in \mathcal{R}_q^2$ with $\mathbf{z} \in \{e^{2i\pi 0/t}, e^{2i\pi 1/t} \ldots, e^{2i\pi(t-1)/t}\}^{N/2}$ and $\|\boldsymbol{\varepsilon}\|_\infty \ll 1/t$.
Output: $\mathsf{ct}_{\mathsf{out}} \in \mathcal{R}_Q^2$.

1 $\mathrm{tmp}_{k-1} \leftarrow \frac{1}{4} + x \cdot \mathsf{PS}(\mathsf{Q_{k,k-1}}, \mathsf{bs}, \mathsf{u}/2, \mathsf{gs}, \mathsf{v})$;
2 $\mathsf{ct}_{k-1} \leftarrow \mathrm{tmp}_{k-1} + \overline{\mathrm{tmp}_{k-1}}$;
3 $bs' \leftarrow \{bs_{2i}\}_i$;
4 $\mathrm{tmp}_{k-2} \leftarrow \frac{1}{4} + x^2 \cdot \mathsf{PS}(\mathsf{Q_{k,k-2}}, \mathsf{bs}', \mathsf{u}/4, \mathsf{gs}, \mathsf{v})$;
5 $\mathsf{ct}_{k-2} \leftarrow \mathrm{tmp}_{k-2} + \overline{\mathrm{tmp}_{k-2}}$;
6 $bs'' \leftarrow \{bs'_{2i}\}_i$;
7 $bs''' = bs'' \cup \{x^u, x^{u+4}, \ldots, x^{2u-4}\}$;
8 $gs' = \{gs_{i+1}\}_i$;
9 $\mathrm{tmp}_{k-3} \leftarrow \frac{1}{4} + x^4 \cdot \mathsf{PS}(\mathsf{Q_{k,k-3}}, \mathsf{bs}''', \mathsf{u}/4, \mathsf{gs}', \mathsf{v}-1)$;
10 $\mathsf{ct}_{k-3} \leftarrow \mathrm{tmp}_{k-3} + \overline{\mathrm{tmp}_{k-3}}$;
11 Repeat Steps 3-10 for the next bits starting from $(u/4, v-1)$;
12 **return** $\mathsf{ct}_{\mathsf{out}}$.

BitExtract is setting the polynomials basis for the Paterson-Stockmeyer polynomial evaluation of $Q_{k,k-1}$. Since it is a polynomial in x^2, only even exponent monomials are computed. Step 1 calls the polynomial evaluation method, abbreviated as PS, taking as arguments the setup basis for baby-steps and giant-steps

and their sizes u and v. Step 2 computes the value of $P_{k,k-1}$ with respect to the formula above. The next polynomial $Q_{k,k-2}$ is smaller. Step 3 is thus halving the bs polynomial basis as explained above. Similarly, Steps 4 and 5 compute $P_{k,k-2}$ from $Q_{k,k-2}$. Step 6 is again halving bs as the algorithm moves to the evaluation of $Q_{k,k-3}$. To balance it with the giant-steps, Step 7 extends the baby-steps basis, which saves a recursive level in giant-steps at Step 8. Steps 9 and 10 evaluate $Q_{k,k-3}$. The next bits are recovered by continually decreasing bs and gs to evaluate $P_{k,\ell}$ with the formula until $\ell = 0$.

6 Experiments

We now describe proof-of-concept implementations, based upon the C++ HEaaN library [Cry22], and report experiment results based on them. These were run on a single-threaded CPU (i.e., Intel Xeon Gold 6242 at 2.8GHz with 502GiB of RAM) running Linux. In the experiments, the variable N denotes the ring degree of the bootstrapping parameter, N_{LWE} denotes the dimension of LWE samples where $N_{\mathsf{LWE}} < N$, h and $\tilde{h}$ denote the Hamming weights of the dense and sparse secret keys respectively (we rely on the sparse secret encapsulation technique from [BTPH22]), $\log_2(QP)$ denotes the maximum switching key modulus, $dnum$ denotes the gadget rank of the gadget decomposition, and $depth$ denotes the remaining multiplicative depths after $\mathsf{IntRootBoot}$. All parameter sets considered in this section reach 128-bit security, according to [APS15]. Note that the security of our new bootstrapping can be analyzed exactly in the same as for conventional CKKS, as we built our scheme upon CKKS without modifying any aspect related to security.

6.1 Bypassing DM/CGGI

As described in Section 4.3, we combined the ring packing from [BCK+23] (i.e., HERMES), the $\mathsf{IntRootBoot}$ algorithm borrowed from [BGGJ20], and our adaptation of the roots-of-unity look-up table of [CKKL24], to bypass DM/CGGI bootstrapping. We have implemented j-bits to j-bits look-up table for $j \in \{2, 4, 6, 8, 10\}$. As the precision for the look-up table interpolation depends on j, we designed optimized parameters for each value of j, as shown in Table 3. Since DM/CGGI is generally effective on processing one or a small number of inputs, we mainly targeted the better latency on small number of inputs. When designing the bootstrapping parameters, we minimized degrees/depths for look-up table evaluations. We also used thin bootstrapping with complex slots, i.e., used fewer slots than available in the ring, to obtain a further latency gain as well as an accuracy improvement.

In Table 4, we batch-evaluated look-up tables for 2^{12} LWE samples of dimension 2^{12}, in all experiments. The method can be generalized to look-up table evaluations on 2^{ℓ} LWE ciphertexts of dimension 2^k. Note that k cannot be too small as a function of the integer bit-length j, as one needs to increase the modulus and dimension to encrypt more bits in a DM/CGGI ciphertext.

	N	$(h, \tilde{h})$	$\log_2(QP)$	$dnum$	$depth$
Param-LUT-2-to-2	2^{14}	$(256, 32)$	440	16	4
Param-LUT-4-to-4	2^{15}		786	4	6
Param-LUT-6-to-6	2^{15}		799	11	8
Param-LUT-8-to-8	2^{15}		869	12	10
Param-LUT-10-to-10	2^{16}		1378	5	12

	$\log_2(q)$					$\log_2(p)$
	Base	StC	Mult and Extract	EvalExp	CtS	
Param-LUT-2-to-2	26	27	26×4	26×8	24×2	27
Param-LUT-4-to-4	33	25×2	33×6	33×8	29×2	61×3
Param-LUT-6-to-6	35	33×2	35×8	35×8	33×2	36×2
Param-LUT-8-to-8	35	33×2	35×10	35×8	33×2	36×2
Param-LUT-10-to-10	42	42×3	42×12	42×8	42×3	61×4

Table 3. Parameters we used for the look-up table implementations. *depth* refers the multiplicative depth after IntRootBoot. The $\log_2(q)$ columns correspond to the primes used for ciphertext modulus, with Base, StC, Mult, Extract, EvalExp and CtS referring to bit-sizes and numbers of primes of the corresponding steps. The column $\log_2(p)$ refers to the bit-sizes and numbers of temporary primes for switching keys. Each parameter set is designed to provide the exact depth required to evaluate an arbitrary look-up table on the required number of bits.

More concretely, our method for batch-evaluating look-up tables on $N_{\mathsf{LWE}} = 2^k$ LWEs of dimension N_{LWE} consists of the following steps:

1. (Ring pack) Map the input LWE ciphertexts into a single N_{LWE}-dimensional RLWE ciphertext. Embed the N_{LWE}-dimensional RLWE ciphertext into an N-dimensional RLWE ciphertext. We consider the N-dimensional RLWE ciphertext as a sparsely packed RLWE ciphertext with $N_{\mathsf{LWE}}/2$ complex slots.
2. (IntRootBoot) Apply IntRootBoot for $N_{\mathsf{LWE}}/2$ out of N slots (i.e., using thin bootstrapping) on the result of the previous step, to get an RLWE ciphertext.
3. (Interpolation) Perform a polynomial interpolation with cleaning functionality.
4. (StC and LWE extraction) Apply thinly-packed StC on the result of the previous step. Decompose the N-dimensional RLWE ciphertext into N_{LWE}-dimensional ciphertexts and extract LWE ciphertexts.

To ensure compatibility with DM/CGGI parameters that have LWE dimensions N_{LWE} that are not 2^k, one can perform pre-processing on the input and post-processing on the output. For the pre-processing, we can simply embed the given LWE ciphertexts into 2^k-dimensional LWE ciphertexts by padding with zeros. For the post-processing, we can switch from dimension 2^k to dimension N_{LWE} LWE as done in CGGI bootstrapping [CGGI16a].

In Table 5, we report experiments for multi-function evaluation (see Section 4.2) in the context of batch-bootstrapping DM/CGGI ciphertexts. We used $\mathsf{Identity}, \mathsf{Square}, \mathsf{Cube}, \mathsf{Backwards} : \{0, \cdots, 2^8 - 1\} \to \{0, \cdots, 2^8 - 1\}$ for the functions, which are defined as $\mathsf{Identity}(x) = x$, $\mathsf{Square}(x) = x^2 \pmod{2^8}$, $\mathsf{Cube}(x) = x^3 \pmod{2^8}$, and $\mathsf{Backwards}(x) = 2^8 - 1 - x$.

Number of bits of the input/output	2	4	6	8	10
$-\log \|\|e\|\|_\infty$	5.4	12.5	12.8	8.7	18.8
HERMES execution time (sec)	0.671	0.717	0.72	0.757	0.76
IntRootBoot execution time (sec)	2.36	4.84	8.09	9.3	18
Interpolation time (sec)	0.09	0.66	1.95	5.1	31.1
StC and LWE extraction time (sec)	0.077	0.19	0.196	0.194	0.39
Total time for look-up table evaluation (sec)	3.2	6.4	11	15.4	50.3
Amortized look-up-table evaluation time (ms)	0.78	1.57	2.67	3.75	12.26

Table 4. Performance evaluation for look-up-tables with various numbers of input/output bits. The second row corresponds to the precision of the integers. The amortized time is computed as total time divided by the number of input LWE ciphertexts. Each reported timings is obtained by averaging over 10 experiments, whereas the precision is maximized over all 10 experiments and all slots.

Number of look-up tables evaluated	1	2	3	4
$-\log \|\|e\|\|_\infty$	8.7			
HERMES execution time (sec)	0.757			
IntRootBoot execution time (sec)	9.3			
Hermite Interpolation time (sec)	5.1	7.1	9.5	10.6
StC and LWE extraction time (sec)	0.194	0.42	0.67	0.77
Total time for look-up table evaluation (sec)	15.4	17.6	20.2	21.4
Amortized look-up-table evaluation time (ms)	3.75	2.14	1.65	1.3

Table 5. Performance evaluation for multi-output look-up tables for various numbers of look-up tables. The second row corresponds to the precision of the integers. We used parameter Param-LUT-8-to-8 to evaluate several 8-bits to 8-bits look-up tables at oncee on the same input. Each reported timings is obtained by averaging over 10 experiments, whereas the precision is maximized over all 10 experiments and all slots.

6.2 Batch Bits Bootstrapping

We instantiated the batch bits bootstrapping (BB-BTS) described in Section 5. For bit extraction, we followed the strategy described in Section 3.3. Below, we report efficiency measurements and provide a comparison to the bits bootstrapping algorithm from [BCKS24]. For a fair comparison, we chose the same ring degree $N = 2^{16}$ and measured the execution time in a similar computing environment. BB-BTS however requires a different CKKS parametrization, which we describe in Table 6. The number of levels reserved for cleaning is estimated as in [BCKS24] (once after 4 sequential gate evaluations).

BB-BTS allows to bootstrap many encrypted bits for the cost of a single (heavier) bootstrapping. In order to quantify the throughput gain, we consider the execution time of BB-BTS and the amount of computation that can be done between two consecutive bootstraps. The amortized gate evaluation time is defined as

$$\frac{T_{\mathsf{BB\text{-}BTS}}}{n \times \ell \times k} \ ,$$

	N	$(h, \tilde{h})$	$\log_2(QP)$	$dnum$	$depth$
Param-BB-BTS	2^{16}	$(256, 32)$	1585	3	24

$\log_2(q)$					$\log_2(p)$
Base	StC	Mult and Extract	EvalExp	CtS	
35	60×1	30×24	35×7	35×3	60×7

Table 6. Parameter we used for batch bits bootstrapping. Here $\log_2(q)$ denotes the primes used for ciphertext modulus, with Base, StC, Mult, Extract, EvalExp and CtS referring to bit-sizes and numbers of primes of the corresponding steps. $\log_2(p)$ refers to the bit-size and number of temporary primes for switching keys.

where $T_{\text{BB-BTS}}$ denotes the BB-BTS time, n denotes the number of slots for bootstrapping, ℓ denotes the number of remaining levels after BB-BTS excluding the levels reserved for cleaning, and k denotes the number of bits. This definition is a generalization of the amortized gate evaluation time used in [BCKS24, Section 5.2]. The detailed results are provided in Table 7. Note that our bit extraction method BitExtract consumes one more multiplicative level for each additional bit. If ℓ_1 denotes the number of useful levels when $k = 1$, then $\ell \approx \ell_1 - k$, and one sees that the term $1/(\ell \times k)$ decreases when k increases, until k reaches approximately $\ell_1/2$. If k further increases, then the amortized cost increases as well, as the number of useful levels becomes too low. This phenomenon limits the speedup factor. For Param-BB-BTS, the speedup factor of BB-BTS on k bits compared to BB-BTS on 1 bit (which is slightly slower than the method in [BCKS24]) is thus bounded from above as $\approx 4.5 = (24 - 6)/4$. This neglects the fact that run-time also grows with k.

Number of ciphertexts (2 bits per slot)	1	2	3	4	5	6	7	8
Number of bits per slot	2	4	6	8	10	12	14	16
Number of levels after bit extraction	23	22	21	19	18	17	16	15
$-\log_2(\|e_{\text{BB-BTS}}\|_\infty)$	12	11	10	9	8	7	6	5
Number of levels reserved for cleaning	6	6	5	5	5	5	5	4
IntRootBoot execution time (sec)	24.0	23.3	23.8	23.8	23.7	23.5	23.6	24.5
Bit extraction time (sec)	0.106	0.862	2.50	4.94	7.80	12.5	20.7	34.3
Amortized gate evaluation time (μs)	21.6	11.5	8.36	7.83	**7.39**	7.62	8.78	10.2

Table 7. Performance evaluation of BB-BTS for the parametrization in Table 6 (i.e., Param-BB-BTS). Here $\|e_{\text{BB-BTS}}\|_\infty$ denotes the maximum error of BB-BTS across all the slots and ciphertexts.

Regarding the amortized gate evaluation time, the experimental optimum is obtained when bootstrapping 10 bits (i.e., 5 ciphertexts) together. In that case, we obtain an amortized gate evaluation time of $7.39\mu s$. Recall that [BCKS24] reached $17.6\mu s$ per binary gate, i.e., our BB-BTS reaches an amortized gate cost that is **2.38**x smaller.

References

ADE+23. E. Aharoni, N. Drucker, G. Ezov, E. Kushnir, H. Shaul, and O. Soceanu. E2E near-standard and practical authenticated transciphering. IACR eprint 2023/1040, 2023.

APS15. M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 2015. Software available at https://github.com/malb/lattice-estimator (commit fd4a460).

BCK+23. Y. Bae, J. H. Cheon, J. Kim, J. H. Park, and D. Stehlé. HERMES: Efficient ring packing using MLWE ciphertexts and application to transciphering. In *CRYPTO*, 2023.

BCKS24. Y. Bae, J. H. Cheon, J. Kim, and D. Stehlé. Bootstrapping bits with CKKS. In *EUROCRYPT*, 2024.

BGGJ20. C. Boura, N. Gama, M. Georgieva, and D. Jetchev. CHIMERA: combining ring-LWE-based fully homomorphic encryption schemes. *J. Math. Cryptol.*, 2020.

BGV12. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS*, 2012.

Bra12. Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *CRYPTO*, 2012.

BTPH22. J.-P. Bossuat, J. Troncoso-Pastoriza, and J.-P. Hubaux. Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In *ACNS*, 2022.

CCS19. H. Chen, I. Chillotti, and Y. Song. Improved bootstrapping for approximate homomorphic encryption. In *EUROCRYPT*, 2019.

CGGI16a. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *ASIACRYPT*, 2016.

CGGI16b. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast fully homomorphic encryption library (version 1.1), 2016. Software available at https://tfhe.github.io/tfhe/.

CGGI17. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In *ASIACRYPT*, 2017.

CHK+18. J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song. Bootstrapping for approximate homomorphic encryption. In *EUROCRYPT*, 2018.

CHMS22. O. Cosseron, C. Hoffmann, P. Méaux, and F.-X. Standaert. Towards case-optimized hybrid homomorphic encryption - featuring the Elisabeth stream cipher. In *ASIACRYPT*, 2022.

CIM19. S. Carpov, M. Izabachène, and V. Mollimard. New techniques for multi-value input homomorphic evaluation and applications. In *CT-RSA*, 2019.

CJP21. I. Chillotti, M. Joye, and P. Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *CSCML*, 2021.

CKK20. J. H. Cheon, D. Kim, and D. Kim. Efficient homomorphic comparison methods with optimal complexity. In *ASIACRYPT*, 2020.

CKKL24. H. Chung, H. Kim, Y.-S. Kim, and Y. Lee. Amortized large look-up table evaluation with multivariate polynomials for homomorphic encryption. IACR eprint 2024/274, 2024.

CKKS17. J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT*, 2017.

Cry22. CryptoLab. HEaaN library, 2022. Available at https://heaan.it/.
DM15. L. Ducas and D. Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In *EUROCRYPT*, 2015.
DMPS24. N. Drucker, G. Moshkowich, T. Pelleg, and H. Shaul. BLEACH: Cleaning errors in discrete computations over CKKS. *J. Cryptol.*, 2024.
FV12. J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. IACR eprint 2012/144, 2012.
HK20. K. Han and D. Ki. Better bootstrapping for approximate homomorphic encryption. In *CT-RSA*, 2020.
KPK+22. S. Kim, M. Park, J. Kim, T. Kim, and C. Min. EvalRound algorithm in CKKS bootstrapping. In *ASIACRYPT*, 2022.
KS22. K. Kluczniak and L. Schild. FDFB: Full domain functional bootstrapping towards practical fully homomorphic encryption. *TCHES*, 2022.
LLK+22. Y. Lee, J.-W. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and H. Kang. High-precision bootstrapping for approximate homomorphic encryption by error variance minimization. In *EUROCRYPT*, 2022.
LLL+21. J.-W. Lee, E. Lee, Y. Lee, Y.-S. Kim, and J.-S. No. High-precision bootstrapping of RNS-CKKS homomorphic encryption using optimal minimax polynomial approximation and inverse sine function. In *EUROCRYPT*, 2021.
LMSS23. C. Lee, S. Min, J. Seo, and Y. Song. Faster TFHE bootstrapping with block binary keys. In *AsiaCCS*, 2023.
LPR10. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, 2010.
LW23. Z. Liu and Y. Wang. Amortized functional bootstrapping in less than 7 ms, with $\widetilde{O}(1)$ polynomial multiplications. In *ASIACRYPT*, 2023.
LW24. Z. Liu and Y. Wang. Relaxed functional bootstrapping: A new perspective on BGV/BFV bootstrapping. IACR eprint 2024/172, 2024.
PS73. M. S. Paterson and L. J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM J Comput*, 1973.
SSTX09. D. Stehlé, R. Steinfeld, K. Tanaka, and K. Xagawa. Efficient public key encryption based on ideal lattices. In *ASIACRYPT*, 2009.
TCBS23. D. Trama, P.-E. Clet, A. Boudguiga, and R. Sirdey. A homomorphic AES evaluation in less than 30 seconds by means of TFHE. In *WAHC*, 2023.
Zam24. Zama. TFHE-rs: A pure rust implementation of the TFHE scheme for boolean and integer arithmetics over encrypted data. (version 0.6.1), 2024. Software available at https://github.com/zama-ai/tfhe-rs.