# Bootstrapping Bits with CKKS

Youngjin Bae[1], Jung Hee Cheon[1,2], Jaehyung Kim[1], and Damien Stehlé[3]

[1] CryptoLab Inc., Seoul, Republic of Korea
{youngjin.bae, jaehyungkim}@cryptolab.co.kr
[2] Seoul National University, Seoul, Republic of Korea
jhcheon@snu.ac.kr
[3] CryptoLab Inc., Lyon, France
damien.stehle@cryptolab.co.kr

**Abstract.** The Cheon–Kim–Kim–Song (CKKS) fully homomorphic encryption scheme is designed to efficiently perform computations on real numbers in an encrypted state. Recently, Drucker *et al* [J. Cryptol.] proposed an efficient strategy to use CKKS in a black-box manner to perform computations on binary data.
In this work, we introduce several CKKS bootstrapping algorithms designed specifically for ciphertexts encoding binary data. Crucially, the new CKKS bootstrapping algorithms enable to bootstrap ciphertexts containing the binary data in the most significant bits. First, this allows to decrease the moduli used in bootstrapping, saving a larger share of the modulus budget for non-bootstrapping operations. In particular, we obtain full-slot bootstrapping in ring degree $2^{14}$ for the first time. Second, the ciphertext format is compatible with the one used in the DM/CGGI fully homomorphic encryption schemes. Interestingly, we may combine our CKKS bootstrapping algorithms for bits with the fast ring packing technique from Bae *et al* [CRYPTO'23]. This leads to a new bootstrapping algorithm for DM/CGGI that outperforms the state-of-the-art approaches when the number of bootstraps to be performed simultaneously is in the low hundreds.

**Keywords:** Fully Homomorphic Encryption · CKKS · Bootstrapping · Binary computations.

## 1 Introduction

Currently competitive Fully Homomorphic Encryption (FHE) schemes include BGV [BGV12] and BFV [Bra12, FV12] which are designed to operate on finite fields, DM [DM15] and CGGI [CGGI16a] which are designed to operate on binary inputs, and CKKS [CKKS17] which focuses on approximations to real and complex numbers. Among them, those relying on RLWE-format ciphertexts [SSTX09, LPR10], namely BGV, BFV and CKKS, provide high throughput thanks to Single-Instruction Multiple-Data (SIMD) computations. In contrast, those based on LWE-format ciphertexts [Reg05], namely DM and CGGI, provide lower latency.

In this work, we focus on the efficiency of homomorphic evaluation of binary circuits. It is usually considered that DM and CGGI are the prime choice for binary computations as their message space is already binary. However, as put forward in [DMPS24], CKKS can be used to perform binary operations by identifying a bit $b \in \{0,1\}$ with the real number $b+\varepsilon$ for some $\varepsilon$ satisfying $|\varepsilon| \ll 1$ and operating on those real numbers to emulate binary gates. Indeed, any binary gate can be implemented as a real-arithmetic circuit of multiplicative depth 1, while preserving the above binary-to-real encoding. For instance, the binary gate evaluation $a \wedge b$ for $a, b \in \{0,1\}$ can be computed as $a \cdot b$, and $a \vee b$ can be computed as $a+b-a \cdot b$. Such homomorphic operations make the error term $\varepsilon$ grow, but it may be reduced by applying the coarse approximation $h_1$ to the step function introduced in [CKK20]. As shown in [DMPS24, Fig. 3], when the parallelism is sufficiently high (e.g., when evaluating a given circuit many times in parallel), CKKS outperforms DM/CGGI. This was further illustrated in [ADE+23], which used the approach from [DMPS24] to homomorphically perform AES-128 decryption of 512KB of data in only 11.5 minutes by running the HEaaN library [Cry22] on a GPU. In contrast, the authors from [TCBS23] relied on the TFHE library [CGGI16b] and the TFHE programmable bootstrapping [CJP21] to decrypt a single AES-128 block in 28 seconds on a 16-thread workstation. While comparing these results is difficult due to heterogeneous computing environments, this indicates that CKKS outperforms DM/CGGI for high-throughput computations. It is noteworthy that the approach from [DMPS24] relies on the CKKS scheme in a black-box manner, and one may then wonder whether the performance of CKKS for homomorphic computations on binary data can be further improved by adapting CKKS to this specific type of computations.

This state of affairs suggests to use a hybrid construction for homomorphically evaluating binary circuits, *à la* CHIMERA [BGGJ20]: the hybrid would rely on DM/CGGI when the circuit is deep and thin (i.e., it is relatively sequential), and on CKKS when the circuit is sufficiently wide (i.e., the computation enjoys heavy parallelism). Unfortunately, the ciphertexts formats from DM/CGGI and [DMPS24] do not seem readily compatible. More concretely, DM/CGGI consider LWE-format ciphertexts $\mathsf{ct} \in \mathbb{Z}_q^{n+1}$ satisfying

$$\mathsf{ct} \cdot \mathsf{sk} \;\approx\; (q/4) \cdot b \bmod q \;, \tag{1}$$

where $\mathsf{sk}$ is the secret key, $b \in \{0,1\}$ is the encrypted bit and the symbol $\approx$ hides an error whose magnitude is small compared to $q/4$. The RLWE-format ciphertexts $\mathsf{ct}$ from [DMPS24] belong to the module $(\mathbb{Z}_q[X]/(X^N+1))^2$ for $N$ a power of 2. In the case of slots-encoding, they satisfy

$$\mathsf{ct} \cdot \mathsf{sk} \;\approx\; \Delta \cdot \mathsf{iDFT}(\mathbf{b}) \bmod q \;, \tag{2}$$

where $\mathsf{sk}$ is the secret key, $\mathbf{b} \in \{0,1\}^{N/2}$ corresponds to $N/2$ bits, $\Delta$ is a scaling factor that is small compared to $q$ but still large compared to the error hidden in the $\approx$ symbol, and $\mathsf{iDFT} : \mathbb{C}^{N/2} \to \mathbb{Z}[X]/(X^N+1)$ refers to (an adaptation of) the inverse Discrete Fourier Transform. In the case of coefficients-encoding, they satisfy

$$\mathsf{ct} \cdot \mathsf{sk} \;\approx\; \Delta \cdot \iota(\mathbf{b}) \bmod q \;, \tag{3}$$

where $\iota(\mathbf{b}) \in \mathbb{Z}[X]/(X^N+1)$ has binary coefficients containing $b$. A RLWE-format ciphertext can be readily viewed as many LWE-format ciphertexts. Going the other way, i.e., converting many LWE-format ciphertexts into a RLWE-format ciphertext, is known as ring packing. This operation has been extensively studied (see [CGGI17, MS18, BGGJ20, CDKS21, LHH+21, BCK+23]). Beyond the formats (LWE or RLWE) of the ciphertexts, the ways the plaintexts are encoded in the ciphertexts seem difficult to reconcile. The encodings of (2) and (3) are compatible via the $\mathsf{CtS}$ (coefficients to slots) and $\mathsf{StC}$ (slots to coefficients) procedures used in CKKS bootstrapping [CHK+18a]. Going from a scaling factor $\Delta \ll q$ as in (3) to a scaling factor equal to $q/4$ as in (1) can be implemented by multiplying the ciphertext by $\approx (q/4)/\Delta$. Going the other way, i.e., decreasing the scaling factor, seems significantly more complex. One way to decrease the scaling factor $\Delta$, or equivalently to increase the modulus while maintaining the scaling factor $\Delta$, is to extract $\iota(b)$ from $\Delta \cdot \iota(b) + q \cdot I$ for some integer polynomial $I$, where the $q \cdot I$ term corresponds to the "mod $q$" operation. The latter is implemented in CKKS bootstrapping by using a polynomial approximation to the sine function, whose degree and hence evaluation cost grow fast when the scaling factor $\Delta$ nears the modulus $q$.

**Main contribution.** We design two bootstrapping algorithms for CKKS ciphertexts whose underlying plaintexts consist of bits: $\mathsf{BinBoot}$ raises the modulus of a single ciphertext encoding a vector of bits, whereas $\mathsf{GateBoot}$ raises the modulus and (SIMD-)evaluates a gate for two ciphertexts encoding vectors of bits. These bootstrapping algorithms allow to obtain lower latency and higher throughput than achieved with the black-box use of CKKS for binary data from [DMPS24]: we conjecture that our approach is preferable to DM/CGGI for homomorphically evaluating binary circuits when the parallel repetition is as low as around 100, and that it outperforms those schemes by close to three order of magnitudes for massively parallel computations. See Table 1. Further, our bootstrapping procedures are compatible with the DM/CGGI ciphertext formats. In fact, combining the efficient ring packing technique from [BCK+23] with $\mathsf{GateBoot}$ leads to an alternative gate bootstrapping for DM/CGGI. Our implementation is advantageous when there are as low as around 200 DM/CGGI gate bootstraps to be performed (for the same gate). See Table 2.

We stress that our implementation is not optimized: its purpose is only to highlight the strength of the proposed bootstrapping techniques. For example, optimizing the RNS arithmetic ([CHK+18b]) to 32-bit moduli as well as the relinearization parameters (see, e.g., [KLSS23]) are likely to give runtime savings by more than a factor 2.

**Technical overview.** Let us first recall the high-level structure of CKKS bootstrapping. We start with a ciphertext $\mathsf{ct}$ in the form of Equation (3) for some small modulus $q = q_0$. At this stage of the discussion, the plaintext $\mathbf{b}$ is not restricted to be binary and can be a real number. The ciphertext $\mathsf{ct}$ is interpreted as a ciphertext modulo a large modulus $Q$, whose inner product with $\mathsf{sk}$ is $\approx \Delta \cdot \iota(\mathbf{b}) + q_0 \cdot I$ for some small-magnitude integer polynomial $I$. This computationally vacuous step is referred to as $\mathsf{ModRaise}$. It is followed by the $\mathsf{CtS}$ step relying

**Table 1.** Throughput comparison with [LMSS23], [CGGI16b] and [DMPS24]. The 'Naive' variant of [DMPS24] is the direct implementation of the method introduced in that work, while the 'Improved' variant is obtained by more efficient placement of cleaning functions and the use of complex bootstrapping (see Section 5.2 for more details). The first two timings are borrowed from [LMSS23] which used a computing environment similar to ours, whereas the last three are obtained with our implementation.

| | | Number of plaintext slots | Amortized bootstrapping time per gate |
|---|---|---|---|
| [LMSS23] | | 1 | 6.49ms |
| [CGGI16b] | | | 10.5ms |
| [DMPS24] | Naive | $2^{15}$ | 92.6$\mu$s |
| | Improved | $2^{16}$ | 27.7$\mu$s |
| This work (BinBoot-Param16) | | | 17.6$\mu$s |

**Table 2.** Comparison between several DM/CGGI bootstrapping implementations. The winning threshold gives the minimal number of gate bootstraps to be performed in parallel for the same gate, for which our implementation becomes preferable. The timing of [DMPS24] is measured as evaluating a gate and bootstrap given two ciphertexts, and using the FGb parameters of [Cry22].

| | Number of plaintext slots | $T_{\mathsf{boot}}$ | Winning threshold | DM/CGGI Compatibility |
|---|---|---|---|---|
| [LMSS23] | 1 | 6.49ms | 262 | YES |
| [CGGI16b] | | 10.5ms | 162 | YES |
| This work | $2^{14}$ | 1.70s | - | YES |
| [DMPS24] | $2^{15}$ | 9.76s | - | NO |

on (an adaptation of) the discrete Fourier transform. Its purpose is to obtain a new ciphertext $\mathsf{ct}'$, whose inner product with $\mathsf{sk}$ is $\approx \mathsf{iDFT}(\Delta \cdot \iota(\mathbf{b}) + q_0 \cdot I) \bmod Q'$ for some $Q'$ lower than $Q$: it has the coefficients of $\Delta \cdot \iota(\mathbf{b}) + q_0 \cdot I$ in the complex slots. At this stage, the ciphertext is in the form of Equation (2), which enables SIMD computations on the underlying data. The goal of the EvalMod step is to remove the $q_0 \cdot I$ term. It achieves the latter by evaluating a polynomial approximation of a proper scaling of the sine function. The key point is that for inputs $x + (2\pi) \cdot I$ for an integer $I$ and a small-magnitude real number $x$, we have $\sin(x + (2\pi) \cdot I) \approx x$. Once the $q_0 \cdot I$ term has been removed, a SIMD arithmetic circuit can be performed on slots. Finally, the StC step reverses the CtS transformation, to obtain a ciphertext in the form of Equation (3) that is ready for another bootstrap.

*Enhanced CKKS-bootstrapping for binary data.* Once we fix the inputs to be in $\{0, 1\}$, we can construct a better approximation for approximate modular reduction (EvalMod). Instead of evaluating the sine function on $x + (2\pi) \cdot I$ for an integer $I$ and a small-magnitude real number $x$ carrying the plaintext data, we use the extrema of a trigonometric function for a binary input mapped

to $b \in \{0,1\} \subseteq \mathbb{R}$:

$$\forall b \in \{0,1\}, \forall I \in \mathbb{Z} : \ \frac{1}{2}\left(1 - \cos\left(b \cdot \pi + I \cdot 2\pi\right)\right) = b \ .$$

The function and its domain of interest are plotted in Figure 1. This choice eliminates the need for $b$ to be small compared to the period, i.e., the need for $\Delta$ to be small compared to $q_0$. As a result, the scaling factor $\Delta$ can be close to $q_0$, which is compatible with the DM/CGGI ciphertext format. This also leads to a significant efficiency gain compared to general-purpose CKKS and its use for binary inputs [DMPS24]. In the latter case, one typically sets $q_0/\Delta \approx 2^{10}$, whereas we can take $q_0/\Delta = 2$: the base modulus $q_0$ can be decreased by almost 10 bits. This significantly reduces modulus consumption during CtS and EvalMod: recall that each multiplication level consumes modulus; those corresponding to CtS and EvalMod have higher modulus consumption as they encode plaintexts that contain the term $q_0 \cdot I$; the bit-size gain for $q_0$ is hence multiplied by the combined multiplicative depth of CtS and EvalMod when we consider the overall modulus consumption. In total, this amounts to more than 100 bits. This gain then allows to consider more levels of computation in a bootstrapping cycle.

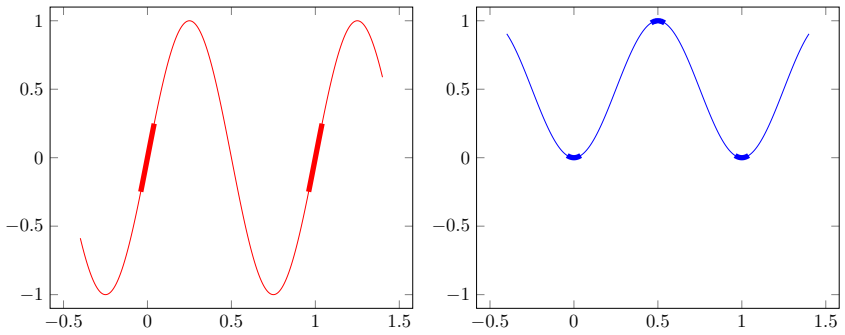


**Fig. 1.** The trigonometric functions used to approximate the modular reduction function, for conventional CKKS (left) and binary bootstrapping (right). The bold areas correspond to valid plaintexts.

Another significant advantage of the modified use of the sine function is that bootstrapping also reduces the error term. As CKKS operates on real numbers, the plaintext is not exactly $b \in \{0,1\}$ but rather $b + \varepsilon$ for some $\varepsilon \ll 1$. In this case, we have

$$\forall b \in \{0,1\}, \forall I \in \mathbb{Z} : \ \frac{1}{2}\left(1 - \cos\left((b+\varepsilon) \cdot \pi + I \cdot 2\pi\right)\right) = b + O(\varepsilon^2) \ .$$

The error shrinks quadratically. This is in contrast with using the sine function for inputs $x$ near 0, which has a linear behaviour even if $x$ encodes a bit. As a result, there is less need to clean the error terms than in [DMPS24].

*CKKS-style gate bootstrapping.* To evaluate a binary gate on two ciphertexts modulo $q_0$, one could run the above binary bootstrapping twice in parallel and then evaluate a binary gate on the bootstrapped ciphertexts as in [DMPS24]. We now propose an alternative CKKS-style gate bootstrapping algorithm inspired from DM/CGGI gate bootstrapping. The objective is to perform most of the work related to the gate on the input small-modulus ciphertexts rather than on high-modulus bootstrapped ciphertexts.

Assume we are given two ciphertexts $\mathsf{ct}_1$ and $\mathsf{ct}_2$ such that $\mathsf{ct}_i \cdot \mathsf{sk} \approx (q/4) \cdot \iota(\mathbf{b}_i) \bmod q_0$ for $i \in \{1,2\}$, where $\iota(\mathbf{b}_i)$ is a binary polynomial containing the coefficients of $\mathbf{b}_i \in \{0,1\}^{N/2}$. We first add the ciphertexts to obtain $\mathsf{ct}$ such that $\mathsf{ct} \cdot \mathsf{sk} \approx (q/4) \cdot \iota(\mathbf{b}_1 + \mathbf{b}_2) \bmod q_0$. Then we note that any symmetric gate $G$ evaluated (SIMD-wise) on $\mathbf{b}_1$ and $\mathbf{b}_2$ is in fact the (SIMD-wise) evaluation of a function of $\mathbf{b}_1 + \mathbf{b}_2$. Importantly, the latter addition occurs over the integers rather than modulo 2. (We observe that $\mathbf{b}_1 + \mathbf{b}_2$ can take only three values and we could hence replace $q/4$ by $q/3$, allowing for a small gain in overall modulus consumption.) The ciphertext $\mathsf{ct}$ then goes through $\mathsf{ModRaise}$ and $\mathsf{CtS}$. The $\mathsf{EvalMod}$ bootstrapping step is modified by changing the sine function for another trigonometric function that allows to send each real $x \in \{0,1,2\}$ to the proper output in $\{0,1\}$ depending on the specific gate. See Figure 3.

The main benefit of the above CKKS-style gate bootstrapping over the binary bootstrapping approach is that one can evaluate gates even at the very bottom level of the multiplication ladder. This is particularly interesting when we are given as inputs LWE ciphertexts with a small modulus, which is likely when we consider the context of ring packing as described in [BCK+23]. This is notably the case if one aims at gate-bootstrapping numerous DM/CGGI ciphertexts in parallel with CKKS. A drawback compared to binary bootstrapping is that it does not contain an error reduction functionality. One can apply the $h_1$ error cleaning function from [CKK20, DMPS24] after evaluating the modified $\mathsf{EvalMod}$, at the cost of two multiplication levels. Alternatively, one can modify $\mathsf{EvalMod}$ further to use three local extrema of a combination of trigonometric functions. The resulting bootstrapping, $\mathsf{GateBoot}'$, can, from this respect, be viewed as an extension of $\mathsf{BinBoot}$ (which relies on two extrema). We refer to the full version of this work for more details.

The design of CKKS-style gate bootstrapping is quite flexible. By relying on trigonometric interpolation, we show that it can handle asymmetric binary gates, gates with more than two binary inputs (such as the majority gate) and gates whose inputs are not binary. The latter corresponds to functional/programmable bootstrapping in the context of DM/CGGI [CJL+20, CLOT21, KS23]. In the full version, we also discuss how to evaluate several gates on the same inputs for a cost that is close to evaluating a single gate, similarly to multi-value bootstrapping in the context of DM/CGGI [CIM19, GBA21].

*Parameter selection and experiments.* In conventional CKKS, parameters are typically set to obtain a relatively high precision (of the order of 20 bits) for each unit homomorphic operation (relinearization, rescaling, etc), in order to achieve sufficient precision at the end of deep real/complex arithmetic circuits.

In case of binary data, we only need to maintain relatively low precision per gate. Indeed, we have a single bit of interest, and we only want to maintain sufficient margin between this bit and the noise resulting from computations. This margin should not be too small, so that the computation can go through with only few noise cleaning steps, either inside binary bootstrapping or based on the $h_1$ function. Evaluating $h_1$ consumes two multiplication levels, and has the effect of (essentially) squaring the error term. On the other hand, the smaller the margin, the smaller the moduli in the modulus chain. This in turns can help to obtain parameters with smaller moduli and ring degree, or to optimize throughput with more non-bootstrapping multiplication levels.

We design two sets of CKKS parameters for binary data. The first one aims at minimizing latency. Based on the moduli optimizations above and GateBoot, we provide the first description of bootstrappable parameters for CKKS with ring degree $N = 2^{14}$. This parameter set handles $2^{13}$ gate bootstrappings at once in less than 1.4s for a single-thread execution on an Intel Xeon Gold 6242 at 2.8GHz with 503GiB of RAM running Linux. It enjoys 2 extra multiplicative levels, which may be used to evaluate second and third binary gates for a negligible cost before another bootstrap is required.

The second parameter set targets high throughput. The ring degree is fixed at $N = 2^{16}$. In 23s with the same computing environment as above, it bootstraps $2^{16}$ bits. The number of available multiplication levels is 28, 8 of which we use for regularly cleaning the error term. This gives an amortized cost per binary gate of $\approx 18\mu$s. This is several hundreds times faster than state-of-the art DM/CGGI bootstrapping [Klu22, BIP+22, LMK+23, LMSS23, XZD+23]. This is also an improvement over [DMPS24] by a factor of the order of 5.3.[4]

**Related works.** The two bootstrapping algorithms introduced above rely on a modification of EvalMod, which approximately evaluates modular reduction with respect to the base modulus $q_0$. This is the most depth-consuming step in CKKS bootstrapping. The use of trigonometric functions has been a typical approach. Initial works [CHK+18a, CCS19, HK20] used trigonometric sine function with Taylor expansion or Chebyshev approximation. In order to reduce the error coming from the difference between the sine and modular reduction functions, approaches based on the inverse sine function [LLL+21] and on the sine series [JM22] have been suggested. Another line of works focuses on directly approximating the modular reduction function. These work rely on Lagrange interpolation [JM20], Least Squares [LLKN20], and Error Variance Minimization [LLK+22]. In our case, we change the function to be evaluated rather than optimize its evaluation.

As seen above, our technique can be viewed as enabling high throughput for DM/CGGI encryption. An independent line of works [MS18, LW23a, LW23b, MKMS23, GPL23] considers the same goal, but by means of modifying the

[4] We note that the $h_1$ noise-reducing function is run after every binary gate in [DMPS24], which is over-conservative as noted in [ADE+23]; a saving of a factor 3.4 can be obtained by calling the $h_1$ function less frequently and using complex bootstrapping.

DM/CGGI bootstrapping algorithms rather than relying on another FHE approach. From a theoretical perspective, this allows to rely on hardness of LWE and RLWE with noise rates polynomially bounded as a function of the LWE dimension and RLWE degree (and a circular security assumption). Timings are only reported in [GPL23] and show that the approach would require further improvements to reach a competitive performance.

One may also consider using the BGV [BGV12] and BFV [Bra12, FV12] schemes to evaluate binary circuits. One approach is to use binary plaintext domains, but, to have SIMD computations, one cannot use the cyclotomic rings $\mathbb{Z}[X]/(X^N+1)$ with degree $N$ that is a power of 2 as the polynomial $X^N+1$ does not factor into $N$ distinct linear terms modulo 2. A possibility is then to switch to more complex cyclotomic rings, as chosen for instance in HELib [HS14]. However, as can be seen in [HS14, Table 3], this approach still does not provide $N$-wise parallelism.[5] Overall, this remains slower than (regular) CKKS bootstrapping [BP23]. Another approach is to keep a power-of-2 cyclotomic ring and use plaintext domain modulo a larger $p$ such that $X^N+1$ factors into $N$ distinct linear terms in order to enjoy $N$-wise parallelism, and only consider plaintext elements in $\{0,1\} \subset \mathbb{Z}_p$. This choice, made for example in Lattigo [EPF22], however makes bootstrapping more complex and less efficient. We note that recent works [KSS24, MHWW24] however show significant progress for bootstrapping performance in this regime. Using BFV for SIMD binary gate evaluations and bootstrapping DM/CGGI has been recently investigated in [LW23c, LW24], though the performance remains limited compared to ours, using CKKS.

## 2 Preliminaries

Given a power-of-two integer $N$, we define the rings $\mathcal{R}_N = \mathbb{Z}[X]/(X^N+1)$ and $\mathcal{R}_{q,N} = \mathcal{R}_N/q\mathcal{R}_N$. Let $\mathsf{DFT} : \mathbb{R}[X]/(X^N+1) \to \mathbb{C}^{N/2}$ be (the variant of) the discrete Fourier transform defined as

$$\forall p \in \mathcal{R}_N : \ \mathsf{DFT}(p) = \left(p(\zeta^{5^i})\right)_{0 \le i < N/2}$$

where $\zeta$ is a primitive $(2N)$-th root of unity. We let $\mathsf{iDFT} : \mathbb{C}^{N/2} \to \mathcal{R}_N$ denote its inverse.

Vectors are denoted in bold. For a vector $\mathbf{b}$, we let $\|\mathbf{b}\|$ denote its 2-norm. This notation is extended to elements of $\mathcal{R}_N$ by first transforming the considered polynomial in the vector of its coefficients. The notation $\mathbf{b} \cdot \mathbf{b}'$ refers to the inner product of the vectors $\mathbf{b}$ and $\mathbf{b}'$ over their ring of definition.

For a function $f : \mathbb{C} \to \mathbb{C}$, we let $f^{\odot}$ denote its component-wise application to a vector over $\mathbb{C}$.

[5] This is actually intrinsic. The number of slots $s$ modulo 2 is exactly the number of distinct factors modulo 2 of the cyclotomic polynomial and the degrees of these factors are all equal. Assume $N$ is the degree of the cyclotomic polynomial, and $d$ the degree of the factors: the number of distinct factors is $s \le N/d$. We also have $s \le 2^d$, the number of polynomials modulo 2 of degree $< d$. The second bound implies that $d \ge \log_2(s)$ and the first one then gives $s \cdot \log_2(s) \le N$.

### 2.1 The CKKS scheme

We first recall some necessary material on the CKKS fully homomorphic encryption scheme [CKKS17, CHK+18a].

**Coefficients and slots.** The Discrete Fourier Transform $\mathsf{DFT}$ is a ring homomorphism sending elements in the ring $\mathcal{R}_N$ (the coefficients space) to complex vectors in $\mathbb{C}^{N/2}$ (the slots space). Importantly, it maps polynomial multiplication to component-wise multiplication.

The decoding map $\mathsf{Dcd} : \mathcal{R}_N \to \mathbb{C}^{N/2}$ is defined as

$$\forall p \in \mathcal{R}_N : \ \mathsf{Dcd}(p) = \frac{1}{\Delta} \cdot \mathsf{DFT}(p) \ ,$$

where $\Delta$ denotes the scaling factor associated to the plaintext polynomial $p$. The encoding map $\mathsf{Ecd} : \mathbb{C}^{N/2} \to \mathcal{R}_N$ is an approximation of its inverse, defined as

$$\forall \mathbf{z} \in \mathbb{C}^{N/2} : \ \mathsf{Ecd}(\mathbf{z}) = \lfloor \Delta \cdot \mathsf{iDFT}(\mathbf{z}) \rceil \ .$$

The data $\mathbf{z} \in \mathbb{C}^{N/2}$ to be computed upon is stored in the slots, and the plaintext polynomial $m$ is $\approx \mathsf{Ecd}(\mathbf{z})$.

Note that $\mathsf{DFT}$ is a scaled 2-norm isometry: it satisfies $\|\mathsf{DFT}(p)\|_2 = \sqrt{N/2} \cdot \|p\|_2$ for all $p \in \mathcal{R}_N$. Therefore, any error produced in $\mathcal{R}_N$ is amplified by factor $\sqrt{N/2}$ in 2-norm when considered in $\mathbb{C}^{N/2}$. This implies that the scaling factor $\Delta$ should be at least $\sqrt{N/2}$ times the desired precision. Looking forward, the scaling factor is the amount by which one has to rescale after each homomorphic multiplication, implying that there must be at least some amount of modulus reserved for each multiplication level even if one aims very low plaintext computation precision.

**Ciphertexts.** A ciphertext $\mathsf{ct} = (b, a) \in \mathcal{R}_{q,N}^2$ decrypting to a (polynomial) plaintext $m \in \mathcal{R}_N$ under a secret key $\mathsf{sk} = (1, s)$ satisfies the following equation over $\mathcal{R}_{q,N}$:

$$\mathsf{ct} \cdot \mathsf{sk} = \mathsf{ct} \cdot (1, s) = b + as = m \ ,$$

where $m \in \mathcal{R}_N$ has small-magnitude coefficients compared to $q$ and may correspond to a desired polynomial up to some small error term. For $\mathbf{z} \in \mathbb{C}^{N/2}$, we write $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}}(\mathbf{z})$ to refer to a ciphertext $\mathsf{ct} \in \mathcal{R}_{q,N}^2$ that decrypts to $\Delta \cdot \mathsf{iDFT}(\mathbf{z})$ under $\mathsf{sk}$.

Given a ciphertext $\mathsf{ct} \in \mathcal{R}_{q,N}^2$ for a key $\mathsf{sk}'$ and an RLWE switching key $\mathsf{swk} \in \mathcal{R}_{qp,N}^2$ from $\mathsf{sk}'$ to $\mathsf{sk}$ for some auxiliary integer $p$, the key switching procedure $\mathsf{KS} : \mathcal{R}_{q,N}^2 \times \mathcal{R}_{qp,N}^2 \to \mathcal{R}_{q,N}^2$ outputs a ciphertext $\mathsf{KS}(\mathsf{ct}, \mathsf{swk})$ decrypting to approximately the same message as $\mathsf{ct}$ but using the new secret key $\mathsf{sk}$.

**Homomorphic operations.** Homomorphic addition/subtraction is performed by adding/subtracting ciphertexts in $\mathcal{R}_{q,N}^2$. The inputs and the output are all defined with respect to the same modulus $q$. The plaintexts are being homomorphically added/subtracted as the decryption equation and the $\mathsf{Ecd}$ function are additive homomorphisms (up to some small error terms).

Homomorphic multiplication proceeds in several steps: tensoring, to multiply the underlying plaintexts; relinearization, to decrease the dimension back to the one of the inputs; and rescaling, to master the growth of the error terms. Homomorphic multiplications are significantly more expensive than homomorphic additions/subtractions as they involve polynomial multiplications. Further, because of rescaling, the output is with respect to a modulus $q/q'$ that is smaller than the modulus $q$ of the inputs. To appropriately handle error growth, one typically sets $q' \approx \Delta$. A multiplication between a ciphertext and a plaintext polynomial can be done similarly but without relinearization.

CKKS also supports homomorphic application of any ring automorphism $\phi : \mathcal{R}_N \to \mathcal{R}_N$. This can be used to move data across slots (i.e., apply a permutation of coordinates over $\mathbb{C}^{N/2}$) and to take the complex conjugate (i.e., apply complex conjugation to a vector of $\mathbb{C}^{N/2}$). The latter is denoted by $\mathsf{conj}$. Ring automorphisms require polynomial multiplications but do not consume modulus.

Because of the modulus consumption of homomorphic multiplication, it is convenient to view an arithmetic circuit in terms of multiplication levels: additions and ring automorphisms do not change the level whereas a multiplication decreases the level by 1. Each level is associated to a modulus.

**Bootstrapping.** As each homomorphic multiplication consumes modulus, one eventually reaches the base modulus $q_0$ after some amount of multiplication depth. The bootstrapping allows to recover the modulus budget: it increases the modulus back to a certain point. The conventional CKKS bootstrapping consists of four steps: $\mathsf{StC}$, $\mathsf{ModRaise}$, $\mathsf{CtS}$, and $\mathsf{EvalMod}$.

$$\mathbf{z} \xrightarrow{\mathsf{StC}} z(x) \xrightarrow{\mathsf{ModRaise}} z(x) + q_0 I(x) \xrightarrow{\mathsf{CtS}} \mathbf{z} + q_0 \mathbf{I} \xrightarrow{\mathsf{EvalMod}} \mathbf{z}$$

- **Slots-to-Coefficients**. Given a ciphertext decrypting to a vector $\mathbf{z}$, $\mathsf{StC}$ converts it to a ciphertext decrypting to a (polynomial) plaintext $z(x)$ whose coefficients are entries of $\mathbf{z}$. It consists in homomorphically multiplying by the $\mathsf{DFT}$ matrix.
- **Modulus Raising**. Given a ciphertext $\mathsf{ct} \in \mathcal{R}_{q_0,N}^2$ at the very smallest modulus, we embed it to $\mathcal{R}_{q,N}^2$ with a large modulus $q$. This introduces a $q_0 I(x)$ term whose coefficients are small-magnitude integer multiples of the base modulus $q_0$.
- **Coefficients-to-Slots**. A ciphertext decrypting to a (polynomial) plaintext $z(x) + q_0 I(x)$ is converted to a ciphertext decrypting a vector $\mathbf{z} + q_0 \mathbf{I}$ whose entries are the coefficients of $z(x) + q_0 I(x)$. It consists in homomorphically multiplying by the $\mathsf{DFT}$ matrix.
- **Modular Reduction**. We homomorphically evaluate the modulo-$q_0$ function in order to remove the $q_0 \mathbf{I}$ term. This is implemented using proper

polynomial approximations such as a combination of sine and inverse sine function [LLL+21] or a direct polynomial approximation built to minimize the error variance [LLK+22].

### 2.2 BLEACH

We now recall a strategy introduced in [DMPS24], called BLEACH, that enables Boolean operations using CKKS. We note that this work also covers other types of discrete data such as small integers, which we do not recall here as we are focusing on binary operations.

The values true and false are respectively identified to 1 and 0. By properly using addition, subtraction and multiplication over the real (or complex) numbers, one can emulate any symmetric binary gate. For instance, the 'and', 'or', and 'xor' gates are respectively obtained as

$$x \wedge y = x \cdot y, \quad x \vee y = x + y - x \cdot y, \quad x \oplus y = (x-y)^2 \ .$$

When performing those operations on approximate inputs $x + \varepsilon_x$ and $y + \varepsilon_y$ with $|\varepsilon_x|, |\varepsilon_y| < 1/4$, the output error has magnitude no more than 5 times the maximum of $|\varepsilon_x|$ and $|\varepsilon_y|$ (see [DMPS24, Le. 2]). After several sequential operations, this error becomes significant and must be decreased. This is achieved by means of a cleanup functionality. Cleanup functions send real values near 0 or 1 closer to 0 or 1, respectively. For instance, the $h_1$ map from [CKK20], defined as $h_1(x) = -2x^3 + 3x^2$ for all $x \in \mathbb{R}$, has a cleaning functionality because $h_1(0) = 0$, $h_1(1) = 1$, and $h_1'(0) = h_1'(1) = 0$.

### 2.3 Modulus engineering

Modulus is a valuable resource in the CKKS scheme: when one runs out of it, then bootstrapping must be performed. Recall that one divides the modulus by an integer at every homomorphic multiplication. More concretely, we consider a top modulus $Q_L$ of the form $Q_L = q_0 \cdot \ldots \cdot q_L$ and, at level $i \in \{0, \ldots, L\}$, the current ciphertext modulus is $Q_i = q_0 \cdot \ldots \cdot q_i$. To provide efficient RNS arithmetic [CHK+18b], the $q_i$'s are chosen co-prime and small enough to fit on a 64-bit machine word [CHK+18b]. To save modulus, state-of-the-art CKKS implementations such as [Cry22, EPF22] use optimizations for the choice of moduli.

A common optimization[6] is to multiply by an integer $c$ before bootstrapping and to divide by $c$ after bootstrapping, as described in Algorithm 1. Note that $c$ is chosen integral to avoid additional modulus consumption due to homomorphic multiplication by an arbitrary scalar. In practice, one typically sets $c$ as a small power of 2, such as $c = 2^4$. For the sake of simplicity, we only describe the idea for bootstrapping real-valued inputs. The input modulus $q$ is the $Q_i$ for the

[6] As far as we are aware of, it has not been formally described so far in an article, but it is used in [EPF22, Cry22].

level $i$ corresponding to the start of StC and the output modulus $Q$ is the $Q_j$ for the level $j \geq i$ reached after EvalMod. In Step 3, the function conj refers to homomorphic complex conjugation.

**Algorithm 1:** Advanced CKKS bootstrapping

**Parameter:** $c \in \mathbb{Z}_{>0}$.
**Input** : $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}}(\mathbf{m}) \in \mathcal{R}_q^2$ where $\mathbf{m} \in [-1,1]^{N/2}$.
**Output** : $\mathsf{ct}_{\mathsf{out}} \in \mathcal{R}_Q^2$.
1 $\mathsf{ct}_1 \leftarrow c \cdot \mathsf{ct}$
2 $\mathsf{ct}_2 \leftarrow \mathsf{CtS} \circ \mathsf{ModRaise} \circ \mathsf{StC}(\mathsf{ct}_1)$
3 $\mathsf{ct}_3 \leftarrow (\mathsf{conj}(\mathsf{ct}_2) + \mathsf{ct}_2)/2$
4 $\mathsf{ct}_{\mathsf{out}} \leftarrow \frac{1}{c} \cdot \mathsf{EvalMod}(\mathsf{ct}_3)$
5 **return** $\mathsf{ct}_{\mathsf{out}}$

The purpose of multiplying by $c$ is to increase the CKKS precision for bootstrapping. Indeed, the errors occurring during bootstrapping are typically larger than those occurring during the levels reserved for useful computations (between EvalMod and StC). Using $c$ allows to balance out these two types of errors. The multiplication of the ciphertexts by $c$ implies that the scaling factors used for the bootstrapping levels are a factor $c$ larger than the others. This in turn leads to taking the base modulus $q_0$ a factor $c$ larger than the non-bootstrapping moduli, as the ratio between the base modulus and its corresponding scaling factor determines the accuracy of the polynomial approximation used for EvalMod and hence its depth consumption and runtime.

Now, we explain how the other $q_i$'s in the moduli chain are chosen. For computation levels that are not part of bootstrapping, they are set to be close to the default scaling factor $\Delta$. One may choose higher or lower moduli for the computations requiring higher or lower precision, respectively, but often the magnitudes of moduli are set to be similar because it is a priori unknown which specific operations are going to be performed. For the bootstrapping levels, the general strategy is to choose the moduli to be as large as the encrypted plaintext polynomials. In StC, the plaintext polynomials have magnitudes $\approx \Delta$, so one first tries to set the modulus near $\Delta$. Similarly, in CtS and EvalMod, the starting point is the estimated magnitude of the $q_0 I(x)$ term added by ModRaise. Given a first trial for a moduli chain, one then fine-tunes it by considering the overall bootstrapping precision. In StC and CtS, one often ends up with moduli that are significantly smaller than the starting point. The main reason is that the scaling factors in StC and CtS are only used to scale up the coefficients of the matrices used to homomorphically evaluate DFT and iDFT, and the induced error is usually the smallest among all errors coming from homomorphic computations.

Explicit examples of moduli chains are provided in [BMTPH21, BCC+22].

## 3 BinBoot: Combined Binary Bootstrap and Clean

In this section, we propose a bootstrapping variant for the case where the plaintext underlying the input ciphertext corresponds to binary data.

### 3.1 Description of BinBoot

In the prior works on CKKS bootstrapping, including [CHK+18a] and [CCS19], a gap of typically $\approx 10$ bits between the base scaling factor $\Delta_0$ and the base modulus $q_0$ is required for bootstrapping. This is because the EvalMod relies on an approximation to the mod-$q_0$ function that is accurate only partially. To be specific, the EvalMod step handles $\Delta_0\mathbf{z} + q_0\mathbf{I}$, the result of CtS, as a message $\Delta_0\mathbf{z}/q_0 + \mathbf{I}$ with scale factor $q_0$. The fact that the approximation to the modular reduction function is accurate only in the vicinity of integer points, leads to the requirement that we have $\|\Delta_0\mathbf{z}/q_0\| \ll 1$, or in other words that we have $\Delta_0 \ll q_0$. The modular reduction function is discontinuous and, as a result, it is not possible to find a low-degree high precision polynomial approximation of it for a large domain. By using a value of $\Delta_0$ that is smaller than $q_0$, one inserts a buffer between $\mathbf{I}$ and the desired output $\Delta_0\mathbf{z}/q_0$. In turn, this enables the use of a polynomial approximation of moderate degree. Decreasing the gap between $\Delta_0$ and $q_0$ would require the use of a polynomial approximation of a much higher degree.

Now, consider the case of a message space restricted to binary vectors, i.e., of the form $\Delta_0\mathbf{z}+q_0\mathbf{I}$ with $\mathbf{z} \in \{0,1\}^{N/2}$ (and $\mathbf{I}$ integral, as above). Although we still need a process of removing the $q_0\mathbf{I}$ term, in this case, it now suffices to use a function that is only required to send $0+q_0\mathbb{Z}$ to 0 and $\Delta_0+q_0\mathbb{Z}$ to 1. This leads considering functions that are 1-periodic (after rescaling by $q_0$), which send $0+\mathbb{Z}$ to 0 and $\Delta_0/q_0+\mathbb{Z}$ to 1, and whose derivatives around those points are moderate in order not to limit error amplification. There are plenty of solutions to these constraints, among which we choose the following:

$$\forall x \in \mathbb{R}: \; f_{\mathsf{BinBoot}}(x) = \frac{1}{2}\left(1 - \cos(2\pi x)\right).$$

The function $f_{\mathsf{BinBoot}}$ is plotted in Figure 2. Beyond satisfying the constraints and enjoying some simplicity, it has two very significant advantages. First, it corresponds to setting $\Delta_0 = q_0/2$. The significantly reduced gap between the scaling factor and the modulus allows to choose a smaller $q_0$ and leads to *significant overall savings in modulus consumption*. Second, as the derivative of $f_{\mathsf{BinBoot}}$ vanishes for $x \in (1/2)\cdot\mathbb{Z}$, applying $f_{\mathsf{BinBoot}}$ *reduces numerical inaccuracy* rather than merely not amplifying it too much.

Algorithm 2 describes BinBoot, the proposed binary bootstrapping method. It takes as input a ciphertext ct modulo $q_0$ and with scaling factor $\Delta_0 = q_0/2$, which decrypts to $\approx \boldsymbol{\varphi} \in \{0,1\}^{N/2}$ under the secret key sk. At Step 1, we have that $\mathsf{ct}' = \mathsf{Enc}_{\mathsf{sk}}((\boldsymbol{\varphi}+\boldsymbol{\varepsilon}_1)/2+\mathbf{I})$ for some small-magnitude integer vector $\mathbf{I} \in \mathbb{Z}^{N/2}$ and some small-magnitude $\boldsymbol{\varepsilon}_1 \in \mathbb{C}^{N/2}$ related to $\boldsymbol{\varepsilon}$ and the precisions used in StC and CtS. Step 2 homomorphically takes the real part of $(\boldsymbol{\varphi}+\boldsymbol{\varepsilon}_1)/2+\mathbf{I}$ to obtain

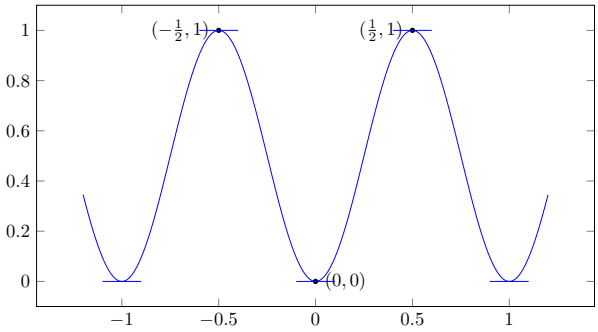


**Fig. 2.** Graph of the trigonometric function $f_{\mathsf{BinBoot}}$ use in Algorithm 2. Note that the derivative vanishes for inputs in $(1/2) \cdot \mathbb{Z}$.

**Algorithm 2:** BinBoot

**Setting:** $\Delta_0 = q_0/2$.
**Input** : $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}}(\boldsymbol{\varphi} + \boldsymbol{\varepsilon}) \in \mathcal{R}_q^2$ with $\boldsymbol{\varphi} \in \{0,1\}^{N/2}$ 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 $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{Eval}_{f_{\mathsf{BinBoot}}}(\mathsf{ct}'')$
4 **return** $\mathsf{ct}_{\mathsf{out}}$

$\mathsf{ct}'' = \mathsf{Enc}_{\mathsf{sk}}((\boldsymbol{\varphi} + \boldsymbol{\varepsilon}_2)/2 + \mathbf{I})$ with $\boldsymbol{\varepsilon}_2$ a small-magnitude vector in $\mathbb{R}^{N/2}$. At Step 3, algorithm $\mathsf{Eval}_{f_{\mathsf{BinBoot}}}$ is the homomorphic evaluation of $f_{\mathsf{BinBoot}}(x) = (1 - \cos(2\pi x))/2$ via appropriate polynomial approximation. By design of $f_{\mathsf{BinBoot}}$ (see also Figure 2), we obtain that $\mathsf{ct}_{\mathsf{out}} = \mathsf{Enc}_{\mathsf{sk}}(\boldsymbol{\varphi} + \boldsymbol{\varepsilon}_{\mathsf{out}})$ for some small-magnitude $\boldsymbol{\varepsilon}_{\mathsf{out}} \in \mathbb{R}^{N/2}$.

### 3.2 Correctness of BinBoot

We start by studying the cleaning functionality of the chosen trigonometric function $f_{\mathsf{BinBoot}}$: the distance of the output to 0 (resp. 1) is essentially the square of the distance of the input to $0 + \mathbb{Z}$ (resp. $1/2 + \mathbb{Z}$) when the latter is sufficiently small. This means that $f_{\mathsf{BinBoot}}$ roughly doubles the precision of the considered data.

**Lemma 1 (Cleaning functionality of $f_{\mathsf{BinBoot}}$).** *Let $\varepsilon \in \mathbb{R}$, $\varphi \in \{0,1\}$ and $I \in \mathbb{Z}$. Then the following holds:*

$$\left| f_{\mathsf{BinBoot}}\left(I + \frac{\varphi + \varepsilon}{2}\right) - \varphi \right| \leq \frac{\pi^2}{4}\varepsilon^2 \ .$$

*Proof.* Observe that

$$f_{\mathsf{BinBoot}}\left(I + \frac{\varphi + \varepsilon}{2}\right) = \frac{1 - \cos((\varphi + \varepsilon)\pi)}{2} = \sin^2\left(\frac{(\varphi + \varepsilon)\pi}{2}\right) \ .$$

We thus have:

$$\left| f_{\mathsf{BinBoot}}\left(I + \frac{\varphi+\varepsilon}{2}\right) - \varphi \right| = \sin^2\left(\frac{\varepsilon\pi}{2}\right) \ ,$$

where, for $\varphi = 1$, we use the identities $\sin(\pi/2+x) = \cos x$ and $\cos^2(x)+\sin^2(x) = 1$ which hold for all $x \in \mathbb{R}$. The proof can be completed by using the inequality $|\sin(x)| \leq |x|$, which also holds for all $x \in \mathbb{R}$. □

We are now ready to state our main theorem on binary bootstrapping.

**Theorem 1 (Binary bootstrapping).** *Consider an execution of* BinBoot *(as defined in Algorithm 2). Take an input ciphertext* $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}}(\boldsymbol{\varphi} + \boldsymbol{\varepsilon})$ *with* $\boldsymbol{\varphi} \in \{0,1\}^{N/2}$ *and* $\boldsymbol{\varepsilon} \in \mathbb{R}^{N/2}$ *such that* $\|\boldsymbol{\varepsilon}\|_\infty \leq B$ *for some* $B$*. Assume that:*

1. *there exist* $B_2$ *and* $B_{\mathbf{I}}$ *such that* $\mathsf{ct}'' = \mathsf{Enc}_{\mathsf{sk}}((\boldsymbol{\varphi}+\boldsymbol{\varepsilon}+\boldsymbol{\varepsilon}_2)/2+\mathbf{I})$ *for some* $\boldsymbol{\varepsilon}_2 \in \mathbb{R}^{N/2}$ *and* $\mathbf{I} \in \mathbb{Z}^{N/2}$ *with* $\|\boldsymbol{\varepsilon}_2\|_\infty \leq B_2$ *and* $\|\mathbf{I}\|_\infty \leq B_{\mathbf{I}}$*;*
2. *there exist* $B_3$ *and* $P_{\mathsf{BinBoot}} \in \mathbb{R}[x]$ *such that* $\mathsf{ct}_{\mathsf{out}} = \mathsf{Enc}_{\mathsf{sk}}(P^{\odot}_{\mathsf{BinBoot}}((\boldsymbol{\varphi} + \boldsymbol{\varepsilon} + \boldsymbol{\varepsilon}_2)/2 + \mathbf{I}) + \boldsymbol{\varepsilon}_3)$ *for some* $\boldsymbol{\varepsilon}_3 \in \mathbb{R}^{N/2}$ *with* $\|\boldsymbol{\varepsilon}_3\|_\infty \leq B_3$ *(recall that* $P^{\odot}_{\mathsf{BinBoot}}$ *refers to the component-wise evaluation of* $P_{\mathsf{BinBoot}}$*);*
3. *there exists* $B_{\mathsf{appr}}$ *such that for all* $x$ *with* $\min(|x|, |x-1/2|) \leq (B+B_2)/2$ *and all integer* $I$ *with* $|I| \leq B_{\mathbf{I}}$*, we have* $|P_{\mathsf{BinBoot}}(x+I) - f_{\mathsf{BinBoot}}(x+I)| \leq B_{\mathsf{appr}}$*.*

*Then we have:*

$$\mathsf{ct}_{\mathsf{out}} = \mathsf{Enc}_{\mathsf{sk}}\left(\boldsymbol{\varphi} + \boldsymbol{\varepsilon}_{\mathsf{out}}\right) \quad \textit{with} \quad \|\boldsymbol{\varepsilon}_{\mathsf{out}}\|_\infty \ \leq \frac{\pi^2}{4}\left(\|\boldsymbol{\varepsilon}\|_\infty + B_2\right)^2 + B_3 + B_{\mathsf{appr}}.$$

*Proof.* By using the assumptions, we obtain that:

$$\mathsf{ct}_{\mathsf{out}} = \mathsf{Enc}_{\mathsf{sk}}\left(f^{\odot}_{\mathsf{BinBoot}}\left(\frac{\boldsymbol{\varphi}+\boldsymbol{\varepsilon}+\boldsymbol{\varepsilon}_2}{2} + \mathbf{I}\right) + \boldsymbol{\varepsilon}_3 + \boldsymbol{\varepsilon}_{\mathsf{appr}}\right) \ ,$$

for some $\boldsymbol{\varepsilon}_2, \boldsymbol{\varepsilon}_3, \boldsymbol{\varepsilon}_{\mathsf{appr}}$ and $\mathbf{I}$ satisfying $\|\boldsymbol{\varepsilon}_2\|_\infty \leq B_2$, $\|\boldsymbol{\varepsilon}_3\|_\infty \leq B_3$, $\|\boldsymbol{\varepsilon}_{\mathsf{appr}}\|_\infty \leq B_{\mathsf{appr}}$ and $\|\mathbf{I}\|_\infty \leq B_{\mathbf{I}}$. Now, Lemma 1 gives that

$$\left\| f^{\odot}_{\mathsf{BinBoot}}\left(\frac{\boldsymbol{\varphi}+\boldsymbol{\varepsilon}+\boldsymbol{\varepsilon}_2}{2} + \mathbf{I}\right) - \boldsymbol{\varphi} \right\|_\infty \leq \frac{\pi^2}{4}(\varepsilon+\varepsilon_2)^2 \ .$$

To complete the proof, it suffices to define:

$$\boldsymbol{\varepsilon}_{\mathsf{out}} = \left(f^{\odot}_{\mathsf{BinBoot}}\left(\frac{\boldsymbol{\varphi}+\boldsymbol{\varepsilon}+\boldsymbol{\varepsilon}_2}{2} + \mathbf{I}\right) - \boldsymbol{\varphi}\right) + \boldsymbol{\varepsilon}_3 + \boldsymbol{\varepsilon}_{\mathsf{appr}} \ . \qquad \square$$

The assumptions may seem cumbersome at first sight, but they merely mean that every usual step of CKKS bootstrapping behaves as expected. Item 1 states that StC, ModRaise, CtS and the homomorphic real part extraction lead to a ciphertext for a plaintext to which an unknown integer vector $\mathbf{I}$ is added as well as a homomorphic computing error $\boldsymbol{\varepsilon}_2$. Note that the size of $\mathbf{I}$ is driven by the size of sk, which is typically chosen ternary (and most often sparse as well). Items 2 and 3 state that the evaluation of $f_{\mathsf{BinBoot}}$ is performed by means

of the evaluation of a polynomial $P_{\mathsf{BinBoot}}$. Item 2 states that the homomorphic evaluation of $P_{\mathsf{BinBoot}}$ induces a small error term $\boldsymbol{\varepsilon}_3$. Item 3 states that $P_{\mathsf{BinBoot}}$ is an accurate approximation of $f_{\mathsf{BinBoot}}$ on the relevant domain. We refer the reader to [CHK+18a] for more details.

By carefully crafting the CKKS moduli chain, it can be arranged that the bootstrapping error bounds $B_2$, $B_3$ and $B_{\mathsf{appr}}$ are all small compared to the maximum allowed value of $\|\boldsymbol{\varepsilon}\|_\infty$.

### 3.3 Modulus engineering for BinBoot

Recall that in the usual EvalMod step of CKKS bootstrapping, the mod-$q_0$ reduction is approximated by a polynomial near integer points only. This is because any polynomial is continuous but modular reduction is not. This leads to setting a gap between the base modulus $q_0$ and the base scaling factor $\Delta_0$ so that $\Delta_0 = \epsilon \cdot q_0$ for a small $\epsilon > 0$. The typical choice of $\epsilon$ is around $2^{-10}$, leading to 10 extra bits of modulus consumption per level during the CtS and EvalMod steps compared to multiplications outside bootstrapping.

BinBoot uses a much smaller gap between $q_0$ and $\Delta_0$, which means less modulus consumption during CtS and EvalMod. Keeping $\Delta_0$ and reducing the size of $q_0$ leads to a reduction in modulus consumption, while maintaining the multiplication precision the same as before. Since CtS and EvalMod are responsible for most of the modulus consumption in bootstrapping, this saves a significant amount of modulus. For example, given a conventional bootstrapping which requires 10 depths in CtS and EvalMod and has 10-bit gap between $q_0$ and $\Delta_0$, using BinBoot allows to save $(10-1) \times 10 = 90$ bits of modulus. This estimate is conservative (to a lesser or larger extent) compared to the data in [BMTPH21, Table 5] and [BCC+22, Tables 6 & 7].

Further, as opposed to a typical CKKS scenario where one aims at real or complex arithmetic with a significant precision of more than 20 bits, here we deal with binary data, i.e., with a single relevant bit. The binary data comes with a noise, inherited from the inaccuracy of the initial encoding and the homomorphic computations. This noise keeps growing during the computations, but it can be reduced with BinBoot (see Theorem 1) or an application of the $h_1$ cleaning map as explained in [DMPS24]. Overall, in terms of precision, we need 1 bit for the binary data, and a few more bits to separate the binary data from the noise. The precision is driven by the default and base scaling factors $\Delta$ and $\Delta_0$, so we may set them smaller than usually done for CKKS. To concretely set $\Delta$ and $\Delta_0$, one should consider the precision loss in each operation and the amount of precision recovery during cleaning. For example, if there are 5 remaining multiplicative depths after bootstrapping, and there is a loss of 1 bit of precision after each binary gate, and if we rely on BinBoot only for cleaning, about 10 bits of precision after bootstrapping could suffice. As a binary gate consumes a single multiplicative depth, there would remain a 5-bit margin between data and noise after the 5 multiplicative levels are exhausted, and this margin would be essentially doubled back to 10 bits thanks to the quadratic noise reduction of BinBoot. A typical choice of $\Delta$ in CKKS is around 40 bits (see, e.g., [BMTPH21, BCC+22])

achieving a bit more than 20 bits of precision. With the 10-bit precision toy example above, it means we can decrease the typical choice of $\Delta$ by 10 bits for binary computations. This saving is multiplied by the total number of levels. We note that this improvement is independent from BinBoot and could be applied to [DMPS24] as well.

The parameters we propose in Section 5 exploit the two improvements described above.

### 3.4 Comparison with BLEACH

The experiments from [DMPS24] relied on conventional CKKS bootstrapping from the HEaaN library [Cry22, version 3.1.4]. In the latter, the most relevant parameters for our discussion are set as $\Delta = 2^{42}$, $\Delta_0 = 2^{45}$ and $q_0 \approx 2^{58}$. This corresponds to the first parameter set of Table 3. Note that $\Delta$ and $\Delta_0$ differ, as Algorithm 1 is used for $c > 1$. We now analyze the effects of both improvements described in Section 3.3.

In the second parameter set of Table 3, we consider keeping the same precision for computations (i.e., keeping the scaling factors $\Delta = 2^{42}$ and $\Delta_0$) but reducing $q_0/\Delta_0$ from $2^{13}$ down to 2. This leads to setting $q_0 \approx 2\Delta_0 = 2^{46}$ instead of $2^{58}$, saving 12 bits of modulus for all levels corresponding to CtS and EvalMod (additionally to the bottom level). While keeping the maximum key switching modulus to be roughly the same, we can increase the available multiplication levels by converting the modulus gain into extra multiplication levels. The new parameter set leads to 13 levels for non-bootstrapping computations compared to 9 levels in the parameter set used in [DMPS24]. Note that our bootstrapping has inherent cleaning functionality but the one in [DMPS24] does not. The cleaning functionality of BinBoot is quadratic, which is equivalent to the one of the $h_1$ map, whose evaluation consumes two multiplicative levels. Assuming that we need exactly one cleaning between two consecutive bootstraps (which is enabled by the high precision provided by large scaling factors), the multiplication depth available for actual computations in a bootstrapping cycle is 7 for [DMPS24] and 13 in our case, i.e., a gain of almost a factor 2. Since the gadget rank *dnum* is fixed and the numbers of moduli are similar in both parameter sets, the bootstrapping performance should be very similar.

When we further optimize the moduli chain by reducing $\Delta$, aiming at 10 bits of precision, we have 29 available multiplication levels outside of bootstrapping. Although we would need more frequent use of cleaning functions, it is still more efficient than the naive approach. For instance, one may clean after every five multiplications, leading to using four $h_1(x) = 3x^2 - 2x^3$ cleaning and 21 remaining levels for binary gate evaluations.

## 4 GateBoot: Combined Bootstrapping and Binary Gate

In this section, we propose an alternative bootstrapping algorithm for binary data that evaluates a binary gate at the same time as it bootstraps.

**Table 3.** Comparison with BLEACH [DMPS24] using concrete parameters. The ring degree is denoted $N$, the largest considered modulus is $\log_2(PQ)$, the total depth is $L$, the number of levels for actual computations (outside of bootstrapping and cleaning) is denoted by *depth*, the key switching gadget rank is denoted by *dnum* (see, e.g., [HK20]) and $\Delta$, $\Delta_0$ and $q_0$ are as in the text. The parameters rely on a ternary secret with Hamming weight 192, and are almost 128 bit secure according to the lattice estimator [APS15]. The 'Proposed - naive' parameter set keeps the same scaling factors as in [DMPS24], whereas the 'Proposed - optimized' parameter set also decreases the precision. In the second table, the '$\log_2(q)$' columns contain the list of bit-sizes of the primes in the moduli chain, split according to their use. Mult corresponds to the non-bootstrapping levels. The '$\log_2(p)$' column contains the list of primes' bit-sizes for the auxiliary moduli used in key switching. The format $X \times Y$ in an entry means that there are $X$ primes of $Y$ bits each.

| Parameter set | $N$ | $\log_2(QP)$ | $L$ | *dnum* | *depth* | $\Delta$ | $\Delta_0$ | $q_0$ |
|---|---|---|---|---|---|---|---|---|
| HEaaN FGb [DMPS24, Cry22] | $2^{16}$ | 1555 | 24 | 5 | **7** | $2^{42}$ | $2^{45}$ | $2^{58}$ |
| Proposed - naive | | 1546 | 28 | 5 | **13** | $2^{42}$ | $2^{45}$ | $2^{46}$ |
| Proposed - optimized | | 1550 | 44 | 5 | **21** | $2^{28}$ | $2^{33}$ | $2^{34}$ |

| Parameter set | $\log_2(q)$ | | | | | $\log_2(p)$ |
|---|---|---|---|---|---|---|
| | Base | StC | Mult | EvalMod | CtS | |
| HEaaN FGb [DMPS24, Cry22] | 58 | $42 \times 3$ | $42 \times 9$ | $58 \times 9$ | $58 \times 3$ | $59 \times 3 + 60 \times 2$ |
| Proposed - naive | 46 | $42 \times 3$ | $42 \times 13$ | $46 \times 9$ | $46 \times 3$ | $46 \times 6$ |
| Proposed - optimized | 34 | $30 \times 2$ | $26 \times 29$ | $34 \times 9$ | $30 \times 3$ | $34 \times 9$ |

### 4.1 Description of GateBoot

Suppose we have two ciphertexts $\mathsf{ct}_1$ and $\mathsf{ct}_2$ that encode binary data $\boldsymbol{\varphi}_1, \boldsymbol{\varphi}_2 \in \{0,1\}^{N/2}$, and that we want to evaluate a symmetric binary gate $G$ (e.g., NAND) in a SIMD manner on $\boldsymbol{\varphi}_1$ and $\boldsymbol{\varphi}_2$. Assume that $\mathsf{ct}_1$ and $\mathsf{ct}_2$ are at the last level before bootstrapping, i.e., they are defined modulo $q$. We could be using BinBoot on both and then evaluate gate $G$ with a degree-1 polynomial in each variable as proposed in [DMPS24].

GateBoot (Algorithm 3) follows a different blueprint. It first adds the two ciphertexts $\mathsf{ct}_1$ and $\mathsf{ct}_2$ before bootstrapping, so that the resulting ciphertext $\mathsf{ct}_{\mathsf{add}}$ decrypts to $\boldsymbol{\varphi}_1 + \boldsymbol{\varphi}_2 \in \{0,1,2\}^{N/2}$. It is important that the addition on the plaintexts is performed over the integers rather than modulo 2 not to lose information. The rationale behind this step is the same as in DM/CGGI: the output of $G$ on $x_1, x_2 \in \{0,1\}$ can be expressed as a function of $x_1 + x_2 \in \{0,1,2\}$, since $G$ is symmetric. As we are considering only ternary vectors at the bottom level, we may set $q_0 = 3\Delta_0$. Note that DM/CGGI usually relies on a power-of-2 ratio rather than a ratio set to 3. We prefer the factor 3 as it provides equally spaced relevant real numbers modulo 1 (namely 0, 1/3 and 2/3), hence allowing to tolerate a slightly higher amount of noise.

Steps 2 and 3 of GateBoot are identical to Steps 1 and 2 of BinBoot. They consist in running StC, ModRaise, CtS and extracting the real parts of the slots. This results in $\mathsf{ct}''$ that contains $(\boldsymbol{\varphi}_1 + \boldsymbol{\varphi}_2 + \boldsymbol{\varepsilon})/3 + \mathbf{I}$ in its slots, for some small-magnitude $\boldsymbol{\varepsilon} \in \mathbb{R}^{N/2}$ and some small-magnitude integer vector $\mathbf{I} \in \mathbb{Z}^{N/2}$.

Step 4 consists in homomorphically evaluating a trigonometric function $f_G$ that removes $\mathbf{I}$ and sends $\boldsymbol{\varphi}_1+\boldsymbol{\varphi}_2$ to $G^{\odot}(\boldsymbol{\varphi}_1, \boldsymbol{\varphi}_2)$. As in conventional CKKS bootstrapping and in $\mathsf{BinBoot}$, it is in fact a polynomial $P_G$ that is being homomorphically evaluated, where $P_G$ is an approximation of the trigonometric function $f_G$. The approximation is required to be accurate for the values of interest, i.e., near $x+I$ for $x$ close to 0, 1/3 or 2/3 and $I$ small. In short, Step 4 clears the period and evaluates the (rest of the) gate simultaneously.

**Algorithm 3:** $\mathsf{GateBoot}_G$ for a symmetric binary gate $G$

**Setting:** $q_0 = 3\Delta_0$.
**Input :** $\mathsf{ct}_i = \mathsf{Enc}_{\mathsf{sk}}(\boldsymbol{\varphi}_i + \boldsymbol{\varepsilon}_i) \in \mathcal{R}_q^2$ for $i = 1, 2$ with $\boldsymbol{\varphi}_i \in \{0,1\}^{N/2}$ and $\|\boldsymbol{\varepsilon}_i\|_\infty \ll 1$.
**Output:** $\mathsf{ct}_{\mathsf{out}} \in \mathcal{R}_Q^2$.

$\mathsf{ct}_{\mathsf{add}} \leftarrow \mathsf{ct}_1 + \mathsf{ct}_2$
$\mathsf{ct}' \leftarrow \mathsf{CtS} \circ \mathsf{ModRaise} \circ \mathsf{StC}(\mathsf{ct}_{\mathsf{add}})$
$\mathsf{ct}'' \leftarrow (\mathsf{conj}(\mathsf{ct}') + \mathsf{ct}')/2$
$\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{Eval}_{f_G}(\mathsf{ct}'')$, with $f_G$ as in Table 4
**return** $\mathsf{ct}_{\mathsf{out}}$

It remains to find trigonometric functions $f_G$ with period 1 such that $f_G((x_1+x_2)/3) = G(x_1, x_2)$ for all $x_1, x_2 \in \{0, 1, 2\}$ and all symmetric binary gates $G$. Functions for the six nontrivial symmetric binary gates are given in Table 4. For example, consider the $\mathsf{NAND}$ gate:

$$\text{if } x_0 = x_1 = 0, \text{ then } x_0 + x_1 = 0 \text{ and } f_{\mathsf{NAND}}(0) = 1 = \mathsf{NAND}(0,0);$$
$$\text{if } x_0 = 1, x_1 = 0, \text{ then } x_0 + x_1 = 1 \text{ and } f_{\mathsf{NAND}}(1/3) = 1 = \mathsf{NAND}(1,0);$$
$$\text{if } x_0 = x_1 = 1, \text{ then } x_0 + x_1 = 2 \text{ and } f_{\mathsf{NAND}}(2/3) = 0 = \mathsf{NAND}(1,1).$$

The graphs of these functions are plotted in Figure 3.

**Table 4.** The trigonometric functions used for the nontrivial symmetric binary gates.

| Gate $G$ | $f_G(x)$ |
|---|---|
| $\mathsf{AND}$ | $\frac{1}{3}\left(1 - 2\sin(2\pi x + \frac{\pi}{6})\right)$ |
| $\mathsf{OR}$ | $\frac{2}{3}\left(1 - \cos(2\pi x)\right)$ |
| $\mathsf{XOR}$ | $\frac{1}{3}\left(1 + 2\sin(2\pi x - \frac{\pi}{6})\right)$ |
| $\mathsf{NAND}$ | $\frac{2}{3}\left(1 + \sin(2\pi x + \frac{\pi}{6})\right)$ |
| $\mathsf{NOR}$ | $\frac{1}{3}\left(1 + 2\cos(2\pi x)\right)$ |
| $\mathsf{XNOR}$ | $\frac{2}{3}\left(1 - \sin(2\pi x - \frac{\pi}{6})\right)$ |

### 4.2 Correctness of GateBoot

Unfortunately, the functions from Table 4 do not have a noise cleaning functionality like $f_{\mathsf{BinBoot}}$ (see Lemma 1). Each function evaluates only one relevant input

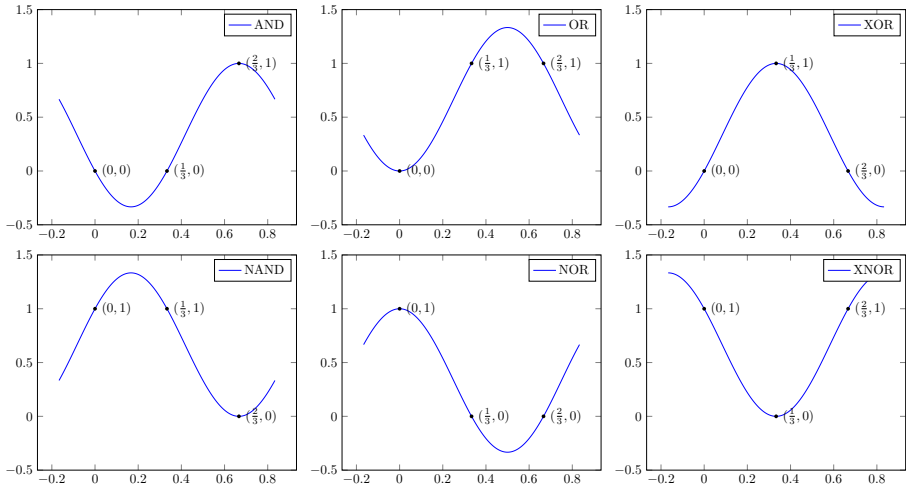


**Fig. 3.** Graphs of $f_{\mathsf{AND}}$, $f_{\mathsf{OR}}$, $f_{\mathsf{XOR}}$, $f_{\mathsf{NAND}}$, $f_{\mathsf{NOR}}$ and $f_{\mathsf{XNOR}}$ used in GateBoot.

(out of three) in a local extremum. If gates are being evaluated on random inputs, then some cleaning occurs in statistical sense, but this property is not easy to exploit. To clean the noise, one may additionally evaluate a noise cleaning function such as $h_1$. Alternatively, we could have chosen period-1 trigonometric functions $f_G$ that have local extrema in 0, 1/3 and 2/3. However, they become more complex and lead to deeper evaluation circuits, and we could not find any advantage of this approach compared to applying $h_1$ after evaluating the functions from Table 4. In the lemma below, we analyze the noise growth incurred by evaluating the functions from Table 4.

**Lemma 2.** *Let $G$ be any nontrivial symmetric binary gate and $f_G : \mathbb{R} \to \mathbb{R}$ as in Table 4. Let $\varepsilon$ be a real number satisfying $|\varepsilon| \leq 1$, $\varphi_1, \varphi_2 \in \{0, 1\}$ and $I \in \mathbb{Z}$. Then, we have:*

$$\left| f_G\left(\frac{\varphi_1 + \varphi_2 + \varepsilon}{3} + I\right) - G(\varphi_1, \varphi_2) \right| \leq \frac{2\sqrt{3}\pi}{9}|\varepsilon| + \frac{2\pi^2}{27}|\varepsilon|^2 .$$

*Proof.* By symmetry of the $f_G$'s, it suffices to prove the result for a single nontrivial symmetric binary gate. We choose $G = \mathsf{NAND}$. Let $\varphi = \varphi_1 + \varphi_2$. Since the $\varphi = 0$ and $\varphi = 1$ cases are symmetric, we only consider the $\varphi = 0$ and $\varphi = 2$ cases.

- Assume that $\varphi = 0$. We must have $\varphi_1 = \varphi_2 = 0$ and $\mathsf{NAND}(\varphi_1, \varphi_2) = 1$. Hence, we have, using the triangle inequality and the fact that the inequality

$|\sin(x)| \leq |x|$ holds for all $x \in \mathbb{R}$:

$$\begin{aligned}
&\left| f_{\mathsf{NAND}}\left(\frac{\varphi_1+\varphi_2+\varepsilon}{3}+I\right) - \mathsf{NAND}(\varphi_1,\varphi_2)\right| \\
&= \frac{1}{3}\left|2\sin\left(\frac{2\pi\varepsilon}{3}+\frac{\pi}{6}\right)-1\right| \\
&= \frac{1}{3}\left|\sqrt{3}\sin\left(\frac{2\pi\varepsilon}{3}\right)+\cos\left(\frac{2\pi\varepsilon}{3}\right)-1\right| \\
&= \frac{1}{3}\left|\sqrt{3}\sin\left(\frac{2\pi\varepsilon}{3}\right)-2\sin^2\left(\frac{\pi\varepsilon}{3}\right)\right| \\
&\leq \frac{\sqrt{3}}{3}\left|\sin\left(\frac{2\pi\varepsilon}{3}\right)\right| + \frac{2}{3}\sin^2\left(\frac{\pi\varepsilon}{3}\right) \\
&\leq \frac{2\sqrt{3}\pi}{9}|\varepsilon| + \frac{2\pi^2}{27}|\varepsilon|^2.
\end{aligned}$$

- Assume that $\varphi = 2$. We must have $\varphi_1 = \varphi_2 = 1$ and $\mathsf{NAND}(\varphi_1,\varphi_2) = 0$. Hence, we have

$$\begin{aligned}
&\left| f_{\mathsf{NAND}}\left(\frac{\varphi_1+\varphi_2+\varepsilon}{3}+I\right) - \mathsf{NAND}(\varphi_1,\varphi_2)\right| \\
&= \frac{2}{3}\left|1+\sin\left(\frac{2\pi\varepsilon}{3}-\frac{\pi}{2}\right)\right| \\
&= \frac{2}{3}\left|1-\cos\left(\frac{2\pi\varepsilon}{3}\right)\right| \\
&= \frac{4}{3}\sin^2\left(\frac{\pi\varepsilon}{3}\right) \\
&\leq \frac{4\pi^2}{27}|\varepsilon|^2 \leq \frac{2\sqrt{3}\pi}{9}|\varepsilon| + \frac{2\pi^2}{27}|\varepsilon|^2.
\end{aligned}$$

  In the last inequality, we used the assumption that $|\varepsilon| \leq 1$.

This completes the proof. □

Using Lemma 2, we can proceed to the main result on GateBoot.

**Theorem 2 (Gate bootstrapping).** *Consider an execution of* GateBoot *(as defined in Algorithm 3) for a nontrivial symmetric binary gate $G$. Take two input ciphertexts $\mathsf{ct}_i = \mathsf{Enc}_{\mathsf{sk}}(\boldsymbol{\varphi}_i + \boldsymbol{\varepsilon}_i)$ with $\boldsymbol{\varphi}_i \in \{0,1\}^{N/2}$ and $\boldsymbol{\varepsilon}_i \in \mathbb{R}^{N/2}$ such that $\|\boldsymbol{\varepsilon}_i\|_\infty \leq B$ for $i \in \{1,2\}$ and some $B$. Let $\boldsymbol{\varphi} = \boldsymbol{\varphi}_1 + \boldsymbol{\varphi}_2 \in \{0,1,2\}^{N/2}$ and $\boldsymbol{\varepsilon} = \boldsymbol{\varepsilon}_1 + \boldsymbol{\varepsilon}_2 \in \mathbb{R}^{N/2}$. Assume that:*

1. *there exist $B_3$ and $B_{\mathbf{I}}$ such that $\mathsf{ct}'' = \mathsf{Enc}_{\mathsf{sk}}((\boldsymbol{\varphi}+\boldsymbol{\varepsilon}+\boldsymbol{\varepsilon}_3)/3+\mathbf{I})$ for some $\boldsymbol{\varepsilon}_3 \in \mathbb{R}^{N/2}$ and $\mathbf{I} \in \mathbb{Z}^{N/2}$ with $\|\boldsymbol{\varepsilon}_3\|_\infty \leq B_3$ and $\|\mathbf{I}\|_\infty \leq B_{\mathbf{I}}$;*
2. *there exist $B_4$ and $P_G \in \mathbb{R}[x]$ such that $\mathsf{ct}_{\mathsf{out}} = \mathsf{Enc}_{\mathsf{sk}}(P_G^{\odot}((\boldsymbol{\varphi}+\boldsymbol{\varepsilon}+\boldsymbol{\varepsilon}_3)/3+\mathbf{I})+\boldsymbol{\varepsilon}_4)$ for some $\boldsymbol{\varepsilon}_4 \in \mathbb{R}^{N/2}$ with $\|\boldsymbol{\varepsilon}_4\|_\infty \leq B_4$;*

3. *there exists* $B_{\mathsf{appr}}$ *such that for all* $x$ *with* $\min(|x|, |x-1/3|, |x-2/3|) \leq (2B+B_3)/3$ *and all integer* $I$ *with* $|I| \leq B_{\mathbf{I}}$*, we have* $|P_G(x+I) - f_G(x+I)| \leq B_{\mathsf{appr}}$.

*Then we have* $\mathsf{ct}_{\mathsf{out}} = \mathsf{Enc}_{\mathsf{sk}}(G^{\odot}(\boldsymbol{\varphi}_1, \boldsymbol{\varphi}_2) + \boldsymbol{\varepsilon}_{\mathsf{out}})$ *with*

$$\|\boldsymbol{\varepsilon}_{\mathsf{out}}\|_\infty \leq \frac{2\sqrt{3}\pi}{9}\left(2\|\boldsymbol{\varepsilon}\|_\infty + B_3\right) + \frac{2\pi^2}{27}\left(2\|\boldsymbol{\varepsilon}\|_\infty + B_3\right)^2 + B_4 + B_{\mathsf{appr}}.$$

*Proof.* By using the assumptions, we obtain that:

$$\mathsf{ct}_{\mathsf{out}} = \mathsf{Enc}_{\mathsf{sk}}\left(f_G^{\odot}\left(\frac{\boldsymbol{\varphi} + \boldsymbol{\varepsilon} + \boldsymbol{\varepsilon}_3}{3} + \mathbf{I}\right) + \boldsymbol{\varepsilon}_4 + \boldsymbol{\varepsilon}_{\mathsf{appr}}\right) ,$$

for some $\boldsymbol{\varepsilon}_3, \boldsymbol{\varepsilon}_4, \boldsymbol{\varepsilon}_{\mathsf{appr}}$ and $\mathbf{I}$ satisfying $\|\boldsymbol{\varepsilon}_3\|_\infty \leq B_2$, $\|\boldsymbol{\varepsilon}_4\|_\infty \leq B_3$, $\|\boldsymbol{\varepsilon}_{\mathsf{appr}}\|_\infty \leq B_{\mathsf{appr}}$ and $\|\mathbf{I}\|_\infty \leq B_{\mathbf{I}}$. Now, Lemma 2 gives that

$$\left\| f_G^{\odot}\left(\frac{\boldsymbol{\varphi} + \boldsymbol{\varepsilon} + \boldsymbol{\varepsilon}_2}{3} + \mathbf{I}\right) - G^{\odot}(\boldsymbol{\varphi}_1, \boldsymbol{\varphi}_2)\right\|_\infty \leq \frac{2\sqrt{3}\pi}{9}\|\boldsymbol{\varepsilon} + \boldsymbol{\varepsilon}_3\|_\infty + \frac{2\pi^2}{27}\|\boldsymbol{\varepsilon} + \boldsymbol{\varepsilon}_3\|_\infty^2 .$$

To complete the proof, it suffices to define:

$$\boldsymbol{\varepsilon}_{\mathsf{out}} = \left(f_G^{\odot}\left(\frac{\boldsymbol{\varphi} + \boldsymbol{\varepsilon} + \boldsymbol{\varepsilon}_3}{3} + \mathbf{I}\right) - G^{\odot}(\boldsymbol{\varphi}_1, \boldsymbol{\varphi}_2)\right) + \boldsymbol{\varepsilon}_4 + \boldsymbol{\varepsilon}_{\mathsf{appr}} . \qquad \square$$

As in Theorem 1, the bootstrapping error bounds $B_3$, $B_4$ and $B_{\mathsf{appr}}$ can all be made small compared to the maximum allowed value $B$ of $\|\boldsymbol{\varepsilon}\|_\infty$.

### 4.3 Comparing GateBoot and BinBoot

Since they rely on similar period-1 trigonometric functions, GateBoot and BinBoot consume approximately the same amount of modulus during bootstrapping. On the one hand, GateBoot evaluates a gate at the same time as it bootstraps, whereas BinBoot does not and would require one extra level to evaluate the gate. On the other hand, BinBoot has an inherent cleaning functionality which is worth two multiplicative depths: for the same cleaning functionality, the GateBoot approach would proceed by evaluating $h_1$, which consumes two levels. Since at least one cleaning is typically required between any two bootstraps, the BinBoot approach may be considered to outperform the GateBoot approach by a multiplicative depth of $2 - 1 = 1$. In the full version of this work, we introduce a variant of GateBoot with cleaning functionality, and provide a detailed comparison between this variant, GateBoot and BinBoot.

Oppositely, when the homomorphic parameters are set small to lower latency, then BinBoot may be over-cleaning compared to the number of gates performed between two consecutive bootstraps. In the GateBoot approach, one would perform cleaning only for a fraction of the bootstrapping cycles. Another context favorable to GateBoot is if we start from LWE-format ciphertexts at the lowest level, as in [BCK+23]. The latter reference contains several motivating applications for storing data in such encryption format. Since GateBoot starts by adding two ciphertexts, it requires only one bootstrap, while the BinBoot approach would require two.

## 5 Experiments

We now present experimental results that showcase the efficiency of the bootstrapping methods proposed in Sections 3 and 4.

When constructing parameters for BGV, BFV and CKKS, a possibility is to choose the smallest ring degree possible to minimize the latency, and another one is to choose a proper ring degree to maximize the throughput. As a certain amount of modulus is necessary for bootstrapping regardless of the ring degree and this amount does not grow very fast with the ring degree, the ring degree for optimizing latency is typically quite suboptimal for throughput. Section 5.1 focuses on low degree to achieve low latency, whereas Section 5.2 uses a larger degree to increase throughput.

Note that even in the case of low latency, we still consider full-slot (real-valued) bootstrapping. One may use a sparsely packed approach [CHK+18a] for minimizing latency and ring degree even more, but we stick to full SIMD computations in order to retain the main advantage of the RLWE-format fully homomorphic encryption schemes. As far as we are aware of, our parameters from Section 5.1 are the first to allow full-slot bootstrapping with ring degree $N = 2^{14}$.

Our implementations are built upon the C++ HEaaN library [Cry22]. The experiments have been conducted single-threaded on an Intel Xeon Gold 6242 at 2.8GHz with 503GiB of RAM running Linux. All the parameters achieve around 128 bits of security according to the lattice estimator [APS15]. We stress that our code is not optimized: its purpose is to highlight the performance of $\mathsf{BinBoot}$ and $\mathsf{GateBoot}$.

The precision is defined as $-\log_2 \|\mathbf{e}\|_\infty$ where $\mathbf{e} \in \mathbb{C}^{N/2}$ is a bootstrapping error vector. More concretely, if $\mathsf{ct} \in \mathcal{R}^2_{Q,N}$ is the ciphertext after bootstrapping, $\mathsf{sk}$ is the secret key and $\mathbf{b}$ is the corresponding plaintext vector of bits, then $\mathbf{e} = \mathsf{Dcd}(\mathsf{ct} \cdot \mathsf{sk}) - \mathbf{b}$. When it is computed for a given experiment, we consider the maximum over 100 samples.

### 5.1 Low latency

Thanks to the reduced modulus consumption, our low latency parameters are for ring degree $N = 2^{14}$. Table 5 outlines the proposed parameter set and its performance. It takes 1.36s and 1.39s for $\mathsf{BinBoot}$ and $\mathsf{GateBoot}$, respectively, for a real full-slot ciphertext (i.e., with $2^{13}$ slots). Bootstrapping precision is 9.6 bits and 7.7 bits for $\mathsf{BinBoot}$ and $\mathsf{GateBoot}$, respectively. We note that the parameter set provides 2 multiplicative depths after bootstrapping.

We compare our results with the state-of-the-art CGGI gate bootstrapping [LMSS23, CGGI16b] to see at which number of LWE ciphertexts our method starts to perform better than CGGI. We borrowed the bootstrapping time results from [LMSS23, Tab. 5] which used Intel(R) i5-12400 at 2.5GHz CPU and 8GB of RAM to measure time. Note that [LMSS23] is based on a novel security assumption, namely LWE with a block binary secret, whereas [CGGI16b] relies on LWE with binary secrets. As shown in Table 6, the fastest CGGI-like implementation takes 6.49ms for a single gate bootstrapping which is 214 times faster

**Table 5.** Description and performance of the small parameter set Param14, designed to lower latency. Here $h$ and $\tilde{h}$ respectively denote the dense and sparse Hamming weights [BTPH22], and $T_{\mathsf{BinBoot}}$ and $P_{\mathsf{BinBoot}}$ (resp. $T_{\mathsf{GateBoot}}$ and $P_{\mathsf{GateBoot}}$) denote the run-time and output precision for BinBoot (resp. GateBoot). The other columns are as in Table 3.

| | $N$ | $(h, \tilde{h})$ | $\log_2(QP)$ | $dnum$ | $depth$ | $T_{\mathsf{BinBoot}}$ | $T_{\mathsf{GateBoot}}$ | $P_{\mathsf{BinBoot}}$ | $P_{\mathsf{GateBoot}}$ |
|---|---|---|---|---|---|---|---|---|---|
| Param14 | $2^{14}$ | $(256, 32)$ | 424 | 13 | 2 | 1.36s | 1.39s | 9.6 | 7.7 |

| $\log_2(q)$ | | | | | $\log_2(p)$ |
|---|---|---|---|---|---|
| Base | StC | Mult | EvalMod | CtS | |
| 32 | 28 | $26 \times 2$ | $32 \times 7$ | $28 \times 2$ | 32 |

than our full slot bootstrapping. In other words, when evaluating at least 214 gates in parallel, our method becomes preferable.

**Table 6.** Comparison with state-of-the-art CGGI gate bootstrapping. The column $T_{\mathsf{boot}}$ contains the bootstrapping times. The timings from the last two rows are borrowed from [LMSS23].

| | $T_{\mathsf{boot}}$ | $T_{\mathsf{GateBoot}}/T_{\mathsf{boot}}$ |
|---|---|---|
| GateBoot-Param14 | 1.39s | 1 |
| [LMSS23] | 6.49ms | 214 |
| [CGGI16b] | 10.5ms | 132 |

Recall that the full number of real slots in our parameter is $2^{13} = 8192$ and we can evaluate two additional gates after bootstrapping using the remaining modulus budget. In addition, we may accelerate the bootstrapping algorithm when we use only a small number of slots by evaluating sparser matrices in StC and CtS. To maintain some generality, we focused on full slots in the comparison although there is some room for optimization.

We may also compare our results with those of [DMPS24]. This work relied on the FGb parameter set of the HEaaN library [Cry22], which takes 9.1s for single bootstrap of a real full-slot ciphertext (with our computing environment). When we directly compare the latency with ours, BinBoot is 6.55 times faster.

### 5.2 High throughput

To optimize throughput, we consider ring degree $N = 2^{16}$. Since bootstrapping can be set to consume the same amount of modulus as for $N = 2^{14}$, the throughput improves as we increase the ring degree. However, larger ring degree leads to larger switching key size and we often want key size to remain sufficiently small. In addition, the throughput increase is no longer significant when we reach certain ring degrees. Our choice of ring degree $N = 2^{16}$ is determined after taking these aspects into account. The parameter set is given in Table 7. BinBoot and GateBoot show slightly worse precision (8.53 bits and 6.61 bits, respectively)

than for Param14. As opposed to Section 5.1, we considered complex full-slot ciphertexts (i.e., with $2^{16}$ slots), to achieve higher throughput.

**Table 7.** Description and performance of the large parameter set Param16, designed to increase throughput. The table columns are as in Tables 3 and 5.

| | $N$ | $(h, \tilde{h})$ | $\log_2(QP)$ | $dnum$ | $depth$ | $T_{\mathsf{BinBoot}}$ | $T_{\mathsf{GateBoot}}$ | $P_{\mathsf{BinBoot}}$ | $P_{\mathsf{GateBoot}}$ |
|---|---|---|---|---|---|---|---|---|---|
| Param16 | $2^{16}$ | $(256, 32)$ | 1598 | 3 | 28 | $23.1s$ | $23.3s$ | 8.53 | 6.61 |

| $\log_2(q)$ | | | | | $\log_2(p)$ |
|---|---|---|---|---|---|
| Base | StC | Mult | EvalMod | CtS | |
| 32 | 32 | $30 \times 28$ | $32 \times 7$ | $32 \times 2$ | $58 \times 7$ |

We now consider the amortized time it takes to evaluate a single gate, by dividing the bootstrapping time with the available depth and the number of slots. Note that we need some cleaning between consecutive bootstrapping cycles in order to maintain precision. Concretely, we expect 1 cleaning step after every 4 (resp. 3) gate evaluations for BinBoot (resp. GateBoot), so we count the number of available levels as $28 - 4 \cdot 2 = 20$ (resp. $28 + 1 - 6 \cdot 2 = 17$). Overall, BinBoot (resp. GateBoot) evaluates a single gate in $17.6\mu$s (resp. $20.9\mu$s), in an amortized sense. Compared to [LMSS23] and [CGGI16b], BinBoot is 369x and 597x faster respectively, as shown in Table 1.[7]

We now compare the performance with [DMPS24]. In the latter work, a cleaning step is performed after every gate, leading to only 3 gate evaluations per bootstrapping cycle. However, since the precision loss is small after each gate, one can reduce the number of cleaning steps greatly, down to only one per bootstrapping cycle. Further, one can use complex bootstrapping instead of real bootstrapping to increase throughput. We compare our results with both the naive and the improved versions of [DMPS24]. The naive (resp. the improved) version evaluates a single gate in $92.6\mu$s (resp. $27.7\mu$s), which is 5.26x (resp. 1.57x) slower than our BinBoot. Note that the runtimes for [DMPS24] are measured using our computing environment, using the HEaaN library [Cry22].

### 5.3 Improving performance further

In principle, if we perform unit operations like Number Theoretic Transform (NTT) on different moduli chains with roughly the same overal bit-size (defined as the bit-size of the product of moduli in the moduli chain), then the run-time should be almost the same. Our new bootstrapping algorithms specific for binary data together with modulus engineering brings significant gain in terms of modulus consumption, which should be converted to performance improvement. For instance in the comparison with [DMPS24] in Section 5.2, the expected performance improvement is roughly by a factor 3 because we have approximately

[7] For the sake of comparison, BinBoot for Param14 and real bootstrapping (optimized for latency), takes $84.8\mu$s per gate which is 4.82x slower than Param16 with complex bootstrapping (optimized for throughput).

three times more multiplicative depths with moduli chains of similar overall bit-sizes. However, our improvement is by a factor 1.57. This is mainly because the modulus gain was not completely converted to a performance improvement. In current RNS implementations, all moduli in a modulus chain are viewed as a 64-bit word, as long as they have fewer than 64 bits. Our moduli are much smaller, but this gain is lost.

We suggest a strategy to overcome this issue, which combines several consecutive rescaling units into a single element in the moduli chain. Observe that NTT only requires the existence of primitive $2N$-th root of unity and each element in the moduli chain needs not be a prime. Therefore, we may combine several consecutive rescaling units (usually primes) into a single modulus which acts as a running modulus for NTT and other modular operations. For example, since most of the primes in the parameter sets of Sections 5.1 and 5.2 have under 32 bits, we may optimize them so that they can be batched by pairs to fit in 64-bit machine words and hence reduce the cost of modular operations by almost a factor 2.[8] The only major difficulty comes from defining a compatible rescaling operation. For this purpose, we may use a conversion from modulo $qq'$ to modulo $q$ (from modulo $\prod_{0\le i\le k} q_i$ to modulo $\prod_{0\le i<k} q_i$, in general) to solve the problem. We leave it as a future work.

## 6 Bootstrapping DM/CGGI Ciphertexts with CKKS

DM/CGGI is more convenient when performing independent operations on bits, and CKKS becomes interesting when there is sufficient parallelism thanks to its support of SIMD computations. For evaluating circuits with heterogeneous amounts of parallelism at different circuit locations, it can be interesting to efficiently switch from one format to the other.

### 6.1 Conversions

Ring packing enables to transform many LWE-format ciphertexts (e.g., DM/CGGI ciphertexts) into a RLWE-format ciphertext. Our work is fully compatible with HERMES [BCK+23], the state-of-the-art ring packing method. First, HERMES performs ring packing at the very bottom of the moduli chain. Second, BinBoot and GateBoot have analogues of the HalfBTS procedure from [CHK+21] used in [BCK+23]. One may replace HalfBTS by HalfBinBoot (Algorithm 2 without CtS and starting at the bottom level) or HalfGateBoot (defined similarly). Third, HalfBinBoot and HalfGateBoot take as inputs ciphertexts that contain the plaintext binary data in their most significant bits. This compatibility provides an alternative bootstrapping approach for DM/CGGI ciphertexts, consisting in running HERMES and then either HalfBinBoot or HalfGateBoot.

[8] One may need $\le 30$ bits primes for lazy modular reductions (see [Har14]). To be compatible with this technique, we may use slightly smaller primes for Param14 and Param16 for Tables 5 and 7.

Going from RLWE-format to LWE-format is relatively simple. We may extract LWE ciphertexts from the bottom-level coefficients-encoded RLWE ciphertexts by selecting and reordering the coefficients, converting a degree-$N$ RLWE ciphertext into $N$ LWE ciphertexts. One may also use key switching and modulus switching to make the dimension and modulus compatible with the desired DM/CGGI format, respectively. If there is a noise bound requirement, then the noise-cleaning functionality in BinBoot or cleaning functions can be used to lower the noise sufficiently before conversion.

### 6.2 Experiments

To experimentally demonstrate this compatibility of formats, we gate-bootstrapped FHEW (i.e., DM) ciphertexts of the OpenFHE library [BBB+22] with GateBoot, implemented with the HEaaN library [Cry22]. Since the FHEW ciphertexts have $q_0 = \Delta_0/4$ (as opposed to our default choice of $q_0 = \Delta_0/3$), we used a slightly modified version of GateBoot whose underlying trigonometric function sends 0, 1/4 and 1/2 to 1, 1 and 0, respectively (we considered the NAND gate). Here 0, 1/4 and 1/2 refer to the data points of interest after adding pairs of FHEW ciphertexts. We then run HERMES and HalfGateBoot to complete the bootstrapping. For HERMES, we used the simplest version from [BCK+23], relying on the column method [HS14] and ring switching [GHPS13].

In the experiment, we used the Param14 parameter set from Table 6, with full-slot complex bootstrapping to evaluate $2^{14}$ gates at once, and the STD128 parameter set for the OpenFHE side. Since we have $(q_0, \Delta_0) = (2^{10}, 2^8)$ in STD128 and $(q_0, \Delta_0) \approx (2^{32}, 2^{31})$ in Param14, we modulus-switched by multiplying (resp. dividing and rounding) by a properly chosen integer to convert LWE ciphertexts from one side to the other. HERMES and HalfGateBoot respectively consume 157ms and 1.54s.

We compared this timing with state-of-the-art CGGI gate-bootstrapping approaches [LMSS23, CGGI16b], in Table 2. Our method becomes favorable compared to [LMSS23] (resp. [CGGI16b]) once the number of gates to be evaluated exceeds 262 (resp. 162).

## Acknowledgement

We thank Elias Suvanto for pointing out an error in Table 3 of a prior version of this work.

## References

[ADE+23] E. Aharoni, N. Drucker, G. Ezov, E. Kushnir, H. Shaul, and O. Soceanu. E2E near-standard and practical authenticated transciphering. Cryptology ePrint Archive, Paper 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).

[BBB+22] A. Al Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, I. Quah, Y. Polyakov, Saraswathy R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca. OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022. Software available at `https://github.com/openfheorg/openfhe-development` (commit 4ebb28e).

[BCC+22] Y. Bae, J. H. Cheon, W. Cho, J. Kim, and T. Kim. META-BTS: Bootstrapping precision beyond the limit. In *CCS*, 2022.

[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.

[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.

[BIP+22] C. Bonte, I. Iliashenko, J. Park, H. V. L. Pereira, and N. P. Smart. FINAL: faster FHE instantiated with NTRU and LWE. In *ASIACRYPT*, 2022.

[BMTPH21] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *EUROCRYPT*, 2021.

[BP23] A. Al Badawi and Y. Polyakov. Demystifying bootstrapping in fully homomorphic encryption. Cryptology ePrint Archive, Paper 2023/149, 2023.

[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.

[CDKS21] H. Chen, W. Dai, M. Kim, and Y. Song. Efficient homomorphic conversion between (ring) LWE ciphertexts. In *ACNS*, 2021.

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

[CHK+18b] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song. A full RNS variant of approximate homomorphic encryption. In *SAC*, 2018.

[CHK+21] J. Cho, J. Ha, S. Kim, B. Lee, J. Lee, J. Lee, S. Moon, and H. Yoon. Transciphering framework for approximate homomorphic encryption. In *ASIACRYPT*, 2021.

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

[CJL+20] I. Chillotti, M. Joye, D. Ligier, J.-B. Orfila, and S. Tap. Concrete: Concrete operates on ciphertexts rapidly by extending TFHE. In *WAHC*, 2020.

[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.

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

[CLOT21] I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In *ASIACRYPT*, 2021.

[Cry22] CryptoLab. HEaaN library, 2022. Available at `https://www.cryptolab.co.kr/en/products-en/heaan-he/`.

[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.

[EPF22] EPFL-LDS, Tune Insight SA. Lattigo v4, 2022. Available at `https://github.com/tuneinsight/lattigo`.

[FV12] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012.

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

[GHPS13] C. Gentry, S. Halevi, C. Peikert, and N. P. Smart. Field switching in BGV-style homomorphic encryption. *J. Comput. Secur.*, 2013.

[GPL23] A. Guimarães, H. V. L. Pereira, and B. Van Leeuwen. Amortized bootstrapping revisited: Simpler, asymptotically-faster, implemented. Cryptology ePrint Archive, Paper 2023/14, 2023.

[Har14] D. Harvey. Faster arithmetic for number-theoretic transforms. *J. of Symb. Comput.*, 2014.

[HK20] K. Han and D. Ki. Better bootstrapping for approximate homomorphic encryption. In *CT-RSA*, 2020.

[HS14] S. Halevi and V. Shoup. Algorithms in HElib. In *CRYPTO*, 2014.

[JM20] C. S. Jutla and N. Manohar. Modular Lagrange interpolation of the mod function for bootstrapping of approximate HE. Cryptology ePrint Archive, Paper 2020/1355, 2020.

[JM22] C. S. Jutla and N. Manohar. Sine series approximation of the mod function for bootstrapping of approximate HE. In *EUROCRYPT*, 2022.

[KLSS23] M. Kim, D. Lee, J. Seo, and Y. Song. Accelerating HE operations from key decomposition technique. In *CRYPTO*, 2023.

[Klu22] K. Kluczniak. NTRU-$\nu$-um: Secure fully homomorphic encryption from NTRU with small modulus. In *CCS*, 2022.

[KS23] K. Kluczniak and L. Schild. FDFB: full domain functional bootstrapping towards practical fully homomorphic encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023.

[KSS24] J. Kim, J. Seo, and Y. Song. Simpler and faster BFV bootstrapping for arbitrary plaintext modulus from CKKS. Cryptology ePrint Archive, Paper 2024/109, 2024.

[LHH+21] W.-J. Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu. PEGASUS: bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *S&P*, 2021.

[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.

[LLKN20] Y. Lee, J.-W. Lee, Y.-S. Kim, and J.-S. No. Near-optimal polynomial for modulus reduction using L2-norm for approximate homomorphic encryption. *IEEE Access*, 2020.

[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.

[LMK+23] Y. Lee, D. Micciancio, A. Kim, R. Choi, M. Deryabin, J. Eom, and D. Yoo. Efficient FHEW bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption. In *EUROCRYPT*, 2023.

[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.

[LW23a] F.-H. Liu and H. Wang. Batch bootstrapping I: A new framework for SIMD bootstrapping in polynomial modulus. In *EUROCRYPT*, 2023.

[LW23b] F.-H. Liu and H. Wang. Batch bootstrapping II: bootstrapping in polynomial modulus only requires $\widetilde{O}(1)$ FHE multiplications in amortization. In *EUROCRYPT*, 2023.

[LW23c] Z. Liu and Y. Wang. Amortized functional bootstrapping inless than 7 ms, with $\tilde{O}(1)$ polynomial multiplications. In *ASIACRYPT*, 2023.

[LW24] Z. Liu and Y. Wang. Relaxed functional bootstrapping: A new perspective on BGV/BFV bootstrapping. Cryptology ePrint Archive, Paper 2024/172, 2024.

[MHWW24] S. Ma, T. Huang, A. Wang, and X. Wang. Accelerating BGV bootstrapping for large $p$ using null polynomials over $\mathbb{Z}_{p^e}$. Cryptology ePrint Archive, Paper 2024/115, to appear in the proceedings of EUROCRYPT'24, 2024.

[MKMS23] G. De Micheli, D. Kim, D. Micciancio, and A. Suhl. Faster amortized FHEW bootstrapping using ring automorphisms. Cryptology ePrint Archive, Paper 2023/112, 2023.

[MS18] D. Micciancio and J. Sorrell. Ring packing and amortized FHEW bootstrapping. In *ICALP*, 2018.

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

[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. At last! A homomorphic AES evaluation in less than 30 seconds by means of TFHE. Cryptology ePrint Archive, Paper 2023/1020, 2023.

[XZD+23] B. Xiang, J. Zhang, Y. Deng, Y. Dai, and D. Feng. Fast blind rotation for bootstrapping FHEs. In *CRYPTO*, 2023.

# A Additional bootstrapping algorithms

In this appendix, we describe GateBoot′, a variant of GateBoot with cleaning functionality. We then provide a comparison between GateBoot′, GateBoot and BinBoot. Finally, we give extensions to more complex setups, and alternative GateBoot functions for XOR and XNOR.

## A.1 GateBoot′: combined bootstrap, gate and clean

As can be observed from the graphs of Figure 3, the chosen functions do not enjoy a cleaning functionality similar to $f_{\mathsf{BinBoot}}$: the derivatives at the points of interest 0, 1/3 and 2/3 are not always vanishing. We now consider more complex functions that correctly implement the six nontrivial binary gates and enjoy vanishing derivatives at the points of interest. The functions are given in Table 8 and plotted in Figure 4. We call GateBoot′ the adaptation of GateBoot (Algorithm 3) to those functions. Lemma 2 and Theorem 2 can be adapted to GateBoot′, with a quadratic error decrease for the function evaluation inherited from the vanishing derivative (as in Lemma 1 and Theorem 1).

**Table 8.** Trigonometric functions used for GateBoot′.

| Gate $G$ | $g_G(x)$ |
|---|---|
| AND | $\frac{1}{3} + \frac{4}{9}\cos(2\pi x + \frac{2\pi}{3}) + \frac{2}{9}\cos(4\pi x - \frac{2\pi}{3})$ |
| OR | $\frac{2}{3} - \frac{4}{9}\cos(2\pi x) - \frac{2}{9}\cos(4\pi x)$ |
| XOR | $\frac{1}{3} + \frac{4}{9}\cos(2\pi x - \frac{2\pi}{3}) + \frac{2}{9}\cos(4\pi x + \frac{2\pi}{3})$ |
| NAND | $\frac{2}{3} - \frac{4}{9}\cos(2\pi x + \frac{2\pi}{3}) - \frac{2}{9}\cos(4\pi x - \frac{2\pi}{3})$ |
| NOR | $\frac{1}{3} + \frac{4}{9}\cos(2\pi x) + \frac{2}{9}\cos(4\pi x)$ |
| XNOR | $\frac{2}{3} - \frac{4}{9}\cos(2\pi x - \frac{2\pi}{3}) - \frac{2}{9}\cos(4\pi x + \frac{2\pi}{3})$ |

**Lemma 3.** *Let $G$ be any nontrivial symmetric binary gate and $g_G : \mathbb{R} \to \mathbb{R}$ as in Table 8. Let $\varepsilon$ be a real number satisfying $|\varepsilon| \leq 1$, $\varphi_1, \varphi_2 \in \{0, 1\}$ and $I \in \mathbb{Z}$. Then, we have:*

$$\left| g_G\left(\frac{\varphi_1 + \varphi_2 + \varepsilon}{3} + I\right) - G(\varphi_1, \varphi_2) \right| \leq \frac{\pi^2}{3}|\varepsilon|^2.$$

*Proof.* The proof technique follows the same format as that of Lemma 2.

By symmetry of the $g_G$'s, it suffices to prove the result for a single nontrivial symmetric binary gate. We choose $G = \mathsf{NAND}$. Let $\varphi = \varphi_1 + \varphi_2$. Since the $\varphi = 0$ and $\varphi = 1$ cases are symmetric, we only consider the $\varphi = 0$ and $\varphi = 2$ cases.

- Assume that $\varphi = 0$. We must have $\varphi_1 = \varphi_2 = 0$ and $\mathsf{NAND}(\varphi_1, \varphi_2) = 1$. Hence, we have, using the triangle inequality and the facts that the inequality

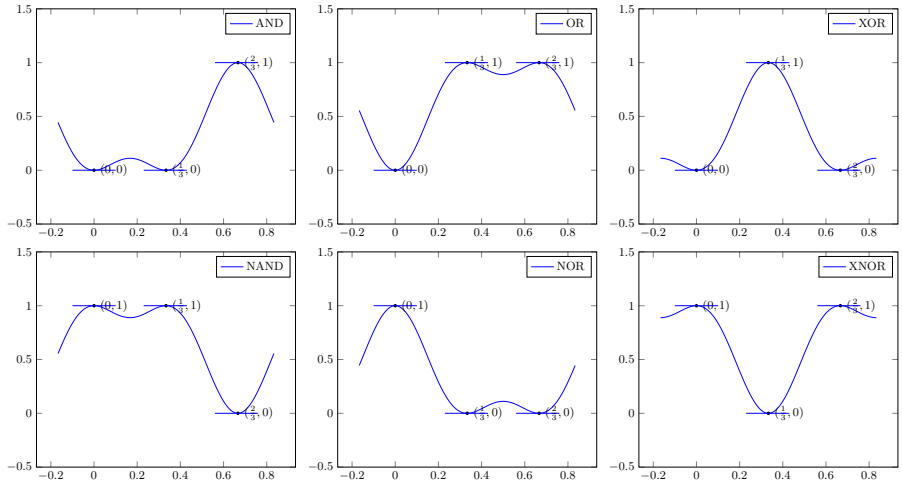


**Fig. 4.** Graphs of $g_{\mathsf{AND}}$, $g_{\mathsf{OR}}$, $g_{\mathsf{XOR}}$, $g_{\mathsf{NAND}}$, $g_{\mathsf{NOR}}$ and $g_{\mathsf{XNOR}}$ used in $\mathsf{GateBoot}'$.

$|\sin(x)| \leq |x|$ holds for all $x \in \mathbb{R}$:

$$
\begin{aligned}
&\left| \; g_{\mathsf{NAND}}\left(\frac{\varphi_1+\varphi_2+\varepsilon}{3}+I\right) - \mathsf{NAND}(\varphi_1,\varphi_2) \; \right| \\
&= \frac{1}{3}\left| -\frac{4}{3}\cos\left(\frac{2\pi\varepsilon+2\pi}{3}\right) - \frac{2}{3}\cos\left(\frac{4\pi\varepsilon-2\pi}{3}\right) - 1 \right| \\
&= \frac{1}{3}\Bigg| -\frac{4}{3}\left(\cos\left(\frac{2\pi\varepsilon}{3}\right)\cos\left(\frac{2\pi}{3}\right) - \sin\left(\frac{2\pi\varepsilon}{3}\right)\sin\left(\frac{2\pi}{3}\right)\right) \\
&\quad -\frac{2}{3}\left(\cos\left(\frac{4\pi\varepsilon}{3}\right)\cos\left(\frac{2\pi}{3}\right) + \sin\left(\frac{4\pi\varepsilon}{3}\right)\sin\left(\frac{2\pi}{3}\right)\right) - 1 \Bigg| \\
&= \frac{1}{3}\left|\frac{1}{3}\left(2\left(\cos\left(\frac{2\pi\varepsilon}{3}\right)-1\right)+\left(\cos\left(\frac{4\pi\varepsilon}{3}\right)-1\right)\right)\right. \\
&\quad \left. -\frac{\sqrt{3}}{3}\left(\sin\left(\frac{4\pi\varepsilon}{3}\right) - 2\sin\left(\frac{2\pi\varepsilon}{3}\right)\right)\right| \\
&\leq \frac{4}{9}\left|\sin^2\left(\frac{\pi\varepsilon}{3}\right)\right| + \frac{2}{9}\left|\sin^2\left(\frac{2\pi\varepsilon}{3}\right)\right| + \frac{2\sqrt{3}}{9}\left|\sin\left(\frac{2\pi\varepsilon}{3}\right)\cdot\left(1-\cos\left(\frac{2\pi\varepsilon}{3}\right)\right)\right| \\
&\leq \frac{4}{9}\left|\sin^2\left(\frac{\pi\varepsilon}{3}\right)\right| + \frac{2}{9}\left|\sin^2\left(\frac{2\pi\varepsilon}{3}\right)\right| + \frac{4\sqrt{3}}{9}\left|\sin\left(\frac{2\pi\varepsilon}{3}\right)\right|\cdot\left|\sin^2\left(\frac{\pi\varepsilon}{3}\right)\right| \\
&\leq \frac{\pi^2}{3}|\varepsilon|^2.
\end{aligned}
$$

In the last inequality, we used the assumption that $|\varepsilon| \leq 1$.

- Assume that $\varphi = 2$. We must have $\varphi_1 = \varphi_2 = 1$ and $\mathsf{NAND}(\varphi_1, \varphi_2) = 0$. Hence, we have

$$\begin{aligned}
&\left| g_{\mathsf{NAND}}\left(\frac{\varphi_1 + \varphi_2 + \varepsilon}{3} + I\right) - \mathsf{NAND}(\varphi_1, \varphi_2)\right| \\
&= \frac{1}{9}\left|-4\cos\left(\frac{2\pi\varepsilon}{3}\right) - 2\cos\left(\frac{4\pi\varepsilon}{3}\right) + 6\right| \\
&\leq \frac{8}{9}\left|\sin^2\left(\frac{\pi\varepsilon}{3}\right)\right| + \frac{4}{9}\left|\sin^2\left(\frac{2\pi\varepsilon}{3}\right)\right| \\
&\leq \frac{\pi^2}{3}|\varepsilon|^2.
\end{aligned}$$

This completes the proof. □

The following is an adaptation of Theorem 2 to $\mathsf{GateBoot}'$. A proof can be obtained by using Lemma 3 rather than Lemma 2 in the proof of Theorem 2.

**Theorem 3 (Gate bootstrapping with cleaning).** *Consider an execution of* $\mathsf{GateBoot}'$ *for a nontrivial symmetric binary gate $G$ (i.e., Algorithm 3 but with $f_G$ replaced by $g_G$). Take two input ciphertexts $\mathsf{ct}_i = \mathsf{Enc}_{\mathsf{sk}}(\boldsymbol{\varphi}_i + \boldsymbol{\varepsilon}_i)$ with $\boldsymbol{\varphi}_i \in \{0,1\}^{N/2}$ and $\boldsymbol{\varepsilon}_i \in \mathbb{R}^{N/2}$ such that $\|\boldsymbol{\varepsilon}_i\|_\infty \leq B$ for $i \in \{1,2\}$ and some $B$. Let $\boldsymbol{\varphi} = \boldsymbol{\varphi}_1 + \boldsymbol{\varphi}_2 \in \{0,1,2\}^{N/2}$ and $\boldsymbol{\varepsilon} = \boldsymbol{\varepsilon}_1 + \boldsymbol{\varepsilon}_2 \in \mathbb{R}^{N/2}$. Assume that:*

1. *there exist $B_3$ and $B_{\mathbf{I}}$ such that $\mathsf{ct}'' = \mathsf{Enc}_{\mathsf{sk}}((\boldsymbol{\varphi}+\boldsymbol{\varepsilon}+\boldsymbol{\varepsilon}_3)/3+\mathbf{I})$ for some $\boldsymbol{\varepsilon}_3 \in \mathbb{R}^{N/2}$ and $\mathbf{I} \in \mathbb{Z}^{N/2}$ with $\|\boldsymbol{\varepsilon}_3\|_\infty \leq B_3$ and $\|\mathbf{I}\|_\infty \leq B_{\mathbf{I}}$;*
2. *there exist $B_4$ and $P_G \in \mathbb{R}[x]$ such that $\mathsf{ct}_{\mathsf{out}} = \mathsf{Enc}_{\mathsf{sk}}(P_G^{\odot}((\boldsymbol{\varphi} + \boldsymbol{\varepsilon} + \boldsymbol{\varepsilon}_3)/3 + \mathbf{I}) + \boldsymbol{\varepsilon}_4)$ for some $\boldsymbol{\varepsilon}_4 \in \mathbb{R}^{N/2}$ with $\|\boldsymbol{\varepsilon}_4\|_\infty \leq B_4$;*
3. *there exists $B_{\mathsf{appr}}$ such that for all $x$ with $\min(|x|, |x-1/3|, |x-2/3|) \leq (2B+B_3)/3$ and all integer $I$ with $|I| \leq B_{\mathbf{I}}$, we have $|P_G(x+I) - f_G(x+I)| \leq B_{\mathsf{appr}}$.*

*Then we have $\mathsf{ct}_{\mathsf{out}} = \mathsf{Enc}_{\mathsf{sk}}(G^{\odot}(\boldsymbol{\varphi}_1, \boldsymbol{\varphi}_2) + \boldsymbol{\varepsilon}_{\mathsf{out}})$ with*

$$\|\boldsymbol{\varepsilon}_{\mathsf{out}}\|_\infty \leq \frac{\pi^2}{3}\left(2\|\boldsymbol{\varepsilon}\|_\infty + B_3\right)^2 + B_4 + B_{\mathsf{appr}}.$$

### A.2 Efficiency aspects

As a preliminary remark, we observe that the modulus engineering techniques described in Section 3.3 for BinBoot are also applicable in the context of GateBoot and $\mathsf{GateBoot}'$. Below, we focus on comparing GateBoot, $\mathsf{GateBoot}'$ and BinBoot.

We saw in Section 4.3 that BinBoot can be considered to outperform GateBoot by 1 multiplicative depth. Still from the perspective of modulus consumption, $\mathsf{GateBoot}'$ is as efficient as BinBoot. As the $g_G$'s functions involve trigonometric functions of period half of those of the $f_G$'s, the polynomials approximating them are expected to be of degrees that are twice higher, leading to an additional depth consumption compared to GateBoot. However, the $g_G$'s enjoy a cleaning functionality, which saves an application of the $h_1$ map, which is worth two levels. Overall, this is equivalent to the BinBoot approach.

Given the above, one option may be better than the other ones depending on the context. Let us assume we want to homomorphically evaluate many SIMD gates. In that case, GateBoot′ is superior to GateBoot. We now consider BinBoot. At first sight, it may seem that the BinBoot approach is more costly than the GateBoot and GateBoot′ approaches: BinBoot seems to require to bootstrap both inputs before applying a gate on them, whereas the other approaches perform an addition at a low level (which has a negligible cost) followed by a single combined bootstrap and gate evaluation. As bootstrapping is costly compared to other operations, this seems to suggest that GateBoot and GateBoot′ are preferable to BinBoot. However, we can consider that the BinBoot approach consumes one more level than GateBoot, to perform a gate *before* running BinBoot rather than *after*. With this perspective, we see that BinBoot has the same cost and same depth consumption as GateBoot′. Note that BinBoot is more flexible than GateBoot′, as it allows easier reuse of ciphertexts.

As seen in Section 4.3, GateBoot can outperform BinBoot when the homomorphic parameters are set small to lower latency, as BinBoot may be over-cleaning. In that regime, GateBoot′ may also be over-cleaning and hence be less interesting than GateBoot.

If we start from LWE-format ciphertexts at the lowest level (as in [BCK+23]), GateBoot′ shares the advantage of GateBoot over BinBoot, as it requires only one bootstrap instead of two. Further, the above solution consisting in applying the gate before bootstrapping does not work here: there is not enough modulus to convert to slot-encoded RLWE-format ciphertexts, perform the gate and apply StC before bootstrapping. Still in the same scenario, GateBoot′ can outperform GateBoot when the LWE-format ciphertexts have large errors, as it cleans right away.

We now consider the scenario of evaluating several gates on the same inputs, similarly to [CIM19] in the case of DM/CGGI. In the case of BinBoot, one can bootstrap two input ciphertexts to place them at a higher multiplication level, and then evaluate whichever gates. For GateBoot (and GateBoot′), Steps 1, 2 and 3 of Algorithm 3 are independent of the gate to be evaluated. We now consider Step 4, i.e., the homomorphic evaluation of an approximation to $f_G$. This step depends on the gate $G$ that one wants to evaluate. However, observe that all the functions listed in Tables 4 and 8 are respectively of the form

$$x \mapsto a + b \cdot \cos(2\pi x + c) \quad \text{and} \quad \mapsto a + b \cdot \cos(2\pi x + c) + d \cdot \cos(4\pi x + e) \ ,$$

for some $a, b, c, d, e \in \mathbb{R}$. Since $\cos(2\pi x + c)$ can be written as a linear combination of $\sin(2\pi x)$ and $\cos(2\pi x)$, one can first evaluate these quantities (which are common to all gates) and then evaluate any $f_G$ from Table 4 by performing homomorphic multiplications by constants and homomorphic additions. This extends to GateBoot′. Depending on the bootstrapping algorithm, the multiplications by constants may consume additional modulus compared to running GateBoot and GateBoot′ for a single gate, showing an advantage of BinBoot.

### A.3 Extensions

We now investigate how to extend gate bootstrapping as described in Algorithm 3 to more complex setups.

Assume that we want to evaluate (in a SIMD manner) a multivariate function

$$C : X_1 \times \ldots \times X_k \to Y$$

where the $X_i$'s and $Y$ are contained in $\mathbb{R}$ and finite with possibly different sizes. We rewrite $C$ as $C = f_C \circ f_{\mathsf{add}}$, such that:

- the function $f_{\mathsf{add}}$ has domain $X_1 \times \ldots \times X_k$ and range some finite $X \subset [0, 1)$, and $f_C$ has domain $X$ and range $Y$;
- the function $f_{\mathsf{add}}$ is of the form $f_{\mathsf{add}}(x_1, \ldots, x_k) = \sum_i (\Delta_i/q)x_i \bmod 1$ for some scaling factors $\Delta_i$;
- the function $f_C$ is 1-periodic;
- it may optionally be required to have a derivative that vanishes over $X$ to provide a cleaning functionality.

For the six nontrivial symmetric binary gates, we took $f_{\mathsf{add}}(x_1, x_2) = (x_1+x_2)/3$, and set $f_G$ as in Table 4. In Appendix A.4, we show that a different choice can be made for XOR and XNOR. More generally, the function $f_C$ can be obtained by trigonometric interpolation: we may look for a function of the form

$$x \;\mapsto\; a_0 + \sum_{j=1}^{\ell} a_j \cdot \cos(2j\pi x) + \sum_{j=1}^{\ell} b_j \cdot \sin(2j\pi x) \;,$$

for some $a_j$'s and $b_j$'s in $\mathbb{R}$ and some integer $\ell$. A solution exists if $2\ell+1 \geq |X|$. To homomorphically evaluate the gate, we proceed as follows: use multiplications by constants and additions at a low level to implement $f_{\mathsf{add}}$; run StC, ModRaise, CtS and real part extraction; homomorphically evaluate a polynomial that closely approximates $f_C$ for inputs of the form $x + \varepsilon + I$ for $x \in X$, $\varepsilon \ll 1$ and a small-magnitude integer $I$.

As a first extension of symmetric binary gates, consider the asymmetric binary gates $G : \{0, 1\} \times \{0, 1\} \to \{0, 1\}$ with $G(0, 1) \neq G(1, 0)$. We may set $f_{\mathsf{add}}(x_1, x_2) = x_1/2 + x_2/4$, which is a bijection. The function $f_G$ can be obtained by trigonometric interpolation. For example, if we consider the gate "$x_1 \leq x_2$", then we may set $f_G(x) = \frac{3}{4} + \frac{1}{2}\cos(2\pi x) - \frac{1}{4}\cos(4\pi x)$.

Taking a bijection and then using a trigonometric interpolation, is always possible. However, it may be costly, as it results in a set $X$ of size $|X| = \prod_i |X_i|$, a complex trigonometric interpolation (i.e., with a large $\ell$), a polynomial approximation of high degree and a deep homomorphic evaluation. One may then aim at minimizing $|X|$. For example, consider the gate that takes three binary inputs and outputs their majority. The default approach results in $|X| = 8$. However, one may observe that the majority is a function of the sum of the three inputs in $X = \{0, 1, 2, 3\}$. Trigonometric interpolation then gives us $f_{\mathsf{MAJ}}(x) = \frac{1}{2}(1-\sqrt{2}\cos(2\pi x - \frac{\pi}{4}))$, which passes through $(0, 0)$, $(1/4, 0)$, $(1/2, 1)$ and $(3/4, 1)$.

### A.4 Alternative GateBoot functions for XOR and XNOR

Below, we describe simpler bootstrap-gate-and-clean functions for XOR and XNOR. For those two gates $G$, the value of $G(x_0, x_1)$ is a function of $x_0+x_1 \bmod 2$ rather than $x_0 + x_1$ over the integers (this is not the case for the remaining four symmetric nontrivial binary gates). We can then set $\Delta_0 = q_0/2$, i.e., set the points of interest as $\mathbb{Z}$ and $1/2 + \mathbb{Z}$, and define:

$$\forall x \in \mathbb{R} : \; h_{\mathsf{XOR}}(x) = \frac{1}{2}\left(1 - \cos(2\pi x)\right) \quad \text{and} \quad h_{\mathsf{XNOR}}(x) = \frac{1}{2}\left(1 + \cos(2\pi x)\right) \; .$$

The graphs of these functions can be found in Figure 5, where it can be seen that the derivatives vanish at the points of interest. However, in a large computation, it may be cumbersome to have a choice of points of interest for XOR and XNOR, and another one for the remaining four symmetric nontrivial binary gates.

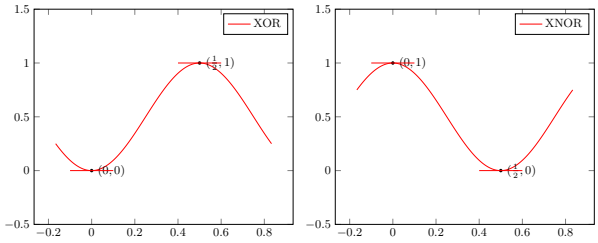


**Fig. 5.** Graphs of alternative choices $h_{\mathsf{XOR}}$ and $h_{\mathsf{XNOR}}$ for GateBoot with $\Delta_0 = q_0/2$.