Evaluating Larger Lookup Tables using CKKS[*][†]

Jules Dumezy[1], Andreea Alexandru[2], Yuriy Polyakov[2], Pierre-Emmanuel Clet[1], Olive Chakraborty[1] and Aymen Boudguiga[1]

[1] Université Paris-Saclay, CEA-List, France
{jules.dumezy,pierre-emmanuel.clet,olive.chakraborty,aymen.boudguiga}@cea.fr
[2] Duality Technologies, USA
{aalexandru,ypolyakov}@dualitytech.com

Abstract.
The Cheon–Kim–Kim–Song (CKKS) scheme is a fully homomorphic encryption scheme that traditionally supports only the evaluation of smooth functions. Recent works have enabled the evaluation of arbitrary (discontinuous) integer functions represented as lookup tables (LUT) on small inputs using the method of functional bootstrapping (FBT). Although well-suited for small integers (up to around 10 bits), the efficiency of FBT quickly declines for large LUTs, and a considerable increase in both runtime and memory requirements is observed. Building on CKKS functional bootstrapping, we propose in this paper two functional bootstrapping algorithms, specifically designed to target larger LUTs (up to 20 bits). For a 16-bit LUT, our implementation in OpenFHE achieves a speed-up of 47.5 in amortized time and 95.1 in latency for single-threaded execution, compared to the state-of-the-art CKKS-based functional bootstrapping method of Alexandru et al. (CRYPTO'25).

Keywords: FHE · Functional Bootstrapping · Programmable Bootstrapping · Lookup Table · CKKS

---

*This work was partly supported by the France 2030 ANR Projects ANR-22-PECY-003 SecureCompute and ANR-23-PECL-0009 TRUSTINCloudS.

†The implementation of our methods is available from the authors upon request for research purposes and is planned for inclusion in the OpenFHE v1.5 release.

Contents

1 Introduction

Gentry's seminal blueprint for fully homomorphic encryption (FHE) showed how to *bootstrap* a somewhat homomorphic encryption scheme into one that can support arbitrary Boolean or arithmetic circuits over encrypted data [Gen09]. For a long time, the natural FHE-supported computations were considered to be circuits, i.e., computations that are expressed as fixed acyclic directed graphs with addition and multiplication nodes. More recent and efficient generations of FHE schemes such as Brakerski-Gentry-Vaikuntanathan (BGV), Brakerski/Fan-Vercauteren (BFV), and Cheon-Kim-Kim-Song (CKKS) [BGV14, Bra12, FV12, CKKS17] would either evaluate such circuits in the leveled mode, or perform periodic bootstrapping to refresh the ciphertexts.

Ducas and Micciancio [DM15] observed that the bootstrapping procedure itself can be generalized: instead of only refreshing the accumulated noise, one can embed a Boolean function into the bootstrapping procedure and evaluate it "for free" during this step. Chillotti et al. [CGGI16] and follow-up works [CGGI20, CJP21] described what is now known as *programmable bootstrapping* (term coined in [CJP21]), where the test polynomial encodes an arbitrary function–represented as a lookup table (LUT)–which is evaluated over the encrypted (binary or small integer) scalar input during the refreshing procedure. In other works, the term *functional bootstrapping* was used [BGGJ19, GBA21, KS22, LMP22, CBSZ23].[1] Another wave of innovations in functional bootstrapping has brought about various solutions to support the evaluation of LUTs over vectors of thousands of inputs simultaneously [MS18, LW23, GPVL23, DMKMS24, BCKS24, BKSS24, AKP24]. In terms of practicality, the vectorized functional bootstrapping solutions built using the CKKS scheme are currently considered as most efficient in terms of amortized runtimes, supporting single-threaded CPU runtimes on the order of one millisecond for 9-bit LUTs and on the order of 10 milliseconds for 14-bit LUTs [AKP24] per slot, i.e., vector component.

These developments allowing the one-shot evaluation of arbitrary functions started pointing towards the (practical) possibility of supporting more general computation models under FHE. An instance of such a more general computation model is the elegant line of work of Hamlin et al. [HHWW19] and Lin et al. [LMW23], which has brought forward theoretical constructions for the random-access memory (RAM) computational model under FHE. In this model, the reads based on encrypted state variables can be obliviously performed (like in ORAM) along with the computation steps over the state under encryption. With the arbitrary function evaluation developments, the boundaries of current practical FHE abilities can be also pushed in the area of attestable encrypted computations, which are of high value in private blockchain transactions [BCMR24]. Computing hash-based signatures or commitments over homomorphically encrypted data for non-polynomial hashes has been the subject of many works, e.g., [BSS$^+$22, WL24], and can benefit from the ability to evaluate under FHE arbitrary functions represented as (large) LUTs.

However, these general computation models cannot be practically supported yet under FHE due to the prohibitively high computational cost of evaluating LUTs over larger input sizes. For instance, the runtime of LUT evaluation using the CKKS functional bootstrapping in [AKP24] becomes substantial when input sizes larger than 14 bits are required (as the 64-bit machine word size is no longer sufficient to support the required CKKS parameters). Therefore, memory address space larger than 14 bits or compression of inputs larger than 14 bits are still practically infeasible. The main goal of our work is to push this limit for evaluating LUTs over vectors of integers further so that larger input spaces could be supported.

[1] Both terms are interchangeable, and we will use the functional bootstrapping term in our work.

1.1 Our contributions

First, we propose a functional bootstrapping procedure that is based on ciphertext bit decomposition and the evaluation of a binary ciphertext multiplexer tree using CKKS. The inspiration for our procedure comes from the most efficient homomorphic LUT evaluation method based on the Chilotti-Gama-Georgieva-Izabachène (CGGI) scheme [CGGI16, BBB$^+$23]. Our procedure, which we refer to as binary multiplexer tree functional bootstrapping (BMT-FBT), is more efficient than the best prior functional bootstrapping method [AKP24], denoted here as Alexandru-Kim-Polyakov functional bootstrapping (AKP-FBT), in terms of throughput starting with 13-bit LUTs, and in terms of latency starting with 10-bit LUTs. Although BMT-FBT deals with a ciphertext multiplexer tree, the building blocks and optimization constraints are very different from the CGGI case. For instance, circuit bootstrapping (the expensive procedure that enables leveled homomorphic multiplications in the CGGI setting), is no longer a bottleneck in our case. The tree evaluation cost in the case of CKKS is determined by the multiplicative depth of the tree rather than the total number of multiplications. We provide several new insights as we develop and optimize the BMT-FBT method.

Observing that the multiplicative depth is the most significant factor determining the efficiency of BMT-FBT, we propose a novel functional bootstrapping method, where the multiplexer tree is collapsed to a smaller depth. The high-level ideas are to use larger digits (instead of bits) and replace the bit decomposition with a specially crafted *functional digit decomposition* procedure, which decomposes digits and evaluates additional LUTs over the digits using the technique of multi-value bootstrapping. We denote this second procedure as collapsed multiplexer tree functional bootstrapping (CMT-FBT). Our comparison of CMT-FBT and BMT-FBT suggests that the former is more efficient for the practical ranges of digit sizes. Moreover, CMT-FBT achieves better throughput than AKP-FBT for LUTs of 11 bits and higher, and better latency starting with 10-bit LUTs. Note that the functional digit decomposition algorithm developed as a building block for CMT-FBT could find other applications and may be of independent interest.

As part of our work, we also introduce several optimizations and extensions for the proposed two methods. We implement both BMT-FBT and CMT-FBT in OpenFHE [AAB$^+$22] and evaluate their performance for single-threaded and multi-threaded settings. Both methods scale linearly with the LUT size, while the prior state-of-the-art method (AKP-FBT) experiences a nonlinear increase in complexity, which makes it already impractical around the LUT size of 14 bits. For 16-bit LUTs (the highest size we were able to run AKP-FBT for), CMT-FBT achieves a two-order-of-magnitude speed-up both for throughput and latency. On a machine with 128 GB of RAM, we were able to evaluate LUTs up to 20 bits in size. To the best of our knowledge, our work provides the first implementation of arbitrary function evaluation using any FHE scheme for LUT sizes up to 20 bits. Although LUTs up to 24 bits were discussed for CGGI in [BBB$^+$23], only theoretical estimates of complexity were provided (without any implementation results).

1.2 Technical overview

Our starting point is AKP-FBT [AKP24]. This method is essentially a hybrid FHE scheme that takes Ring-LWE (RLWE) ciphertexts [LPR10], transforms them to CKKS (using cheap scaling operations), evaluates an LUT using a trigonometric Hermite interpolation (a generalization of CKKS bootstrapping), and then converts the result back to RLWE (again using cheap scaling operations). This scheme can be viewed as a vectorized instantiation of the Ducas-Micciancio (DM) [DM15] blueprint that was previously used for the DM and CGGI hybrid FHE schemes [DM15, CGGI16]. The vectorized method enables functional bootstrapping for many thousands of encrypted numbers at the same time, in contrast to encrypted scalars in the case of DM/CGGI. Although the input scheme looks like BFV, no support for homomorphic multiplications is needed as all the "hard work" is done by CKKS; this is why we refer to the input scheme as RLWE [LPR10], which was used as the base scheme for BFV in [FV12].

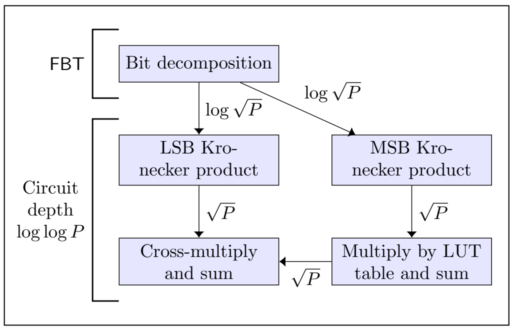
The main bottleneck in AKP-FBT is the evaluation of the power series for the trigonometric Hermite interpolation. To compute an LUT for encrypted numbers in $\mathbb{Z}_P$, one has to evaluate a polynomial of degree P, which becomes the main practical limitation in supporting larger LUTs. The highest value of P practically achieved in [AKP24] and any prior work on functional bootstrapping was 2^{14}. The main goal of our work is to push this limit higher (the methods we propose can evaluate LUTs of up to 2^{20} on a commodity server machine with 128 GB of RAM).

The problem of supporting larger LUT sizes also arises in the more established functional bootstrapping methods based on DM/CGGI (though there the practical limit of the underlying bootstrapping is typically 8 bits [TBC+25]). The two best methods to support larger LUTs for arbitrary functions are tree-based [GBA21] and CMux-based [BBB+23]. Hence, we first examined whether these methods could be adapted to the vectorized case. The tree-based method was already discussed in Section 5.3 of [AKP24]: its main limitation is that encrypted LUTs, which are needed for all but the first level of the tree, are much costlier than plaintext LUTs. More concretely, the plaintext LUTs for a plaintext modulus P can be evaluated in $O(\sqrt{P})$ CKKS ciphertext multiplications while the encrypted LUTs require $O(P)$ CKKS ciphertext multiplications. The second method, based on a ciphertext multiplexer tree, is more promising, but the design constraints in the CKKS setting are very different from the DM/CGGI case.

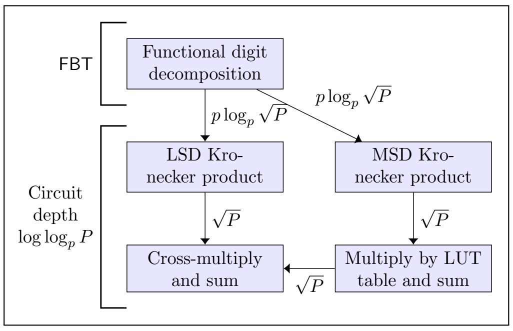
In the DM/CGGI case, the ciphertext is first decomposed using functional bootstrapping into (R)LWE ciphertexts encrypting its bits, then the (R)LWE ciphertexts are extended to Ring GSW (RGSW) to enable RGSW multiplications using a very expensive circuit bootstrapping procedure, and, finally, the multiplexer tree is evaluated using cheap, low-noise RGSW multiplications. In the CKKS case, the "circuit bootstrapping" is relatively cheap (we simply need to add multiplicative levels to CKKS). However, the multiplications themselves are more expensive both in terms of computational complexity and noise growth, and they should be evaluated using the binary tree multiplication method. Moreover, it is important to evaluate $O(P)$ scalar ciphertext multiplications and additions as close to the end of the multiplexer tree computation as possible, to minimize the ciphertext modulus size used during these operations.

With these guidelines in mind, we developed our BMT-FBT method, which is illustrated in Figure 1a. First, we perform the bit decomposition using AKP-FBT. Next, the bits are grouped into vectors: the least significant (LSB) and most significant (MSB) ones. We formulate the products required to evaluate the multiplexer tree as Kronecker products. The Kronecker products are computed for each LSB and MSB separately, using the binary tree method to minimize the consumed ciphertext modulus in CKKS. Then, the ciphertexts resulting from the MSB Kronecker product are multiplied by the LUT table of size P, and aggregated. Finally, the resulting ciphertexts are cross-multiplied by the ciphertexts computed via the LSB Kronecker product and summed up. This algorithm achieves three different optimization objectives specific to CKKS: (1) the multiplicative depth of the multiplexer tree evaluation is set to the minimum possible value of $\log\log P$, (2) the number of ciphertext multiplications is reduced to optimal $O(\sqrt{P})$, and (3) the $O(P)$ scalar ciphertext multiplications and additions are performed at the smallest possible ciphertext modulus.

The main factor determining the efficiency of BMT-FBT for a significant range of LUT sizes is the multiplicative depth required for evaluating the multiplexer tree because (1) a higher depth requirement increases the complexity of bit decomposition (more CKKS levels are needed after the bootstrapping), and (2) the depth increases the complexity of evaluating the multiplexer tree itself. We observe that the depth can be decreased if



(a) Schematic for binary multiplexer tree



(b) Schematic for collapsed multiplexer tree

Figure 1: Workflow diagrams for our functional bootstrapping methods. We show the number of ciphertexts returned by each block next to the arrows. A single ciphertext is input and returned by each procedure.

instead of bits we work with larger digits and then replace the bit decomposition with a more involved method, which we call *functional digit decomposition*. The functional digit decomposition is a novel technique that extracts digits while also applying LUTs on them.

Based on these observations, we propose our second method, CMT-FBT, using vectors of the least significant digits (LSD) and most significant digits (MSD). The CMT-FBT method is illustrated at a high level in Figure 1b. Its main distinction from BMT-FBT is that this method first decomposes the ciphertexts into digits of size p while also evaluating additional LUTs that will be needed during the Kronecker products, which corresponds to the functional digit decomposition procedure. As a result, the multiplicative depth of the multiplexer tree is reduced from $\log\log P$ to $\log\log_p P$. Although this distinction seems straightforward, accurately analyzing the cost requires looking at more than just the number of operations (as in a typical computational complexity analysis), but also at the number of remaining residue number system (RNS) limbs over which these operations occur. We compare BMT-FBT and CMT-FBT and find that CMT-FBT always outperforms BMT-FBT for digit sizes $2 \leq \log p \leq 5$ in a single-threaded setting. Another way to interpret this conclusion is that BMT-FBT can be viewed as a special case of CMT-FBT at $p = 2$ (where functional digit decomposition is replaced with a much simpler bit decomposition procedure and LSD/MSD Kronecker products degenerate to LSB/MSB Kronecker products), and the optimal value of p is larger than two for all practical scenarios.

We also apply a number of optimizations to both BMT-FBT and CMT-FBT, such as lazy relinarization and lazy rescaling for CKKS ciphertext multiplications, implementing loop parallelization, and finding optimal values for the number of digits in CMT-FBT. Moreover, we discuss various extensions of both methods, such as the evaluation of complex-valued LUTs, slot-independent LUT evaluation, and encrypted LUT evaluation.

1.3 Related work

A comparison of our experimental results with state-of-the-art methods (for LUT sizes of 10 bits and above) is summarized in Table 1. We next discuss the state-of-the-art results in more detail.

The idea of evaluating LUTs based on functional bootstrapping first appeared in the context of DM/CGGI schemes. There are three main methods: (1) direct LUT evaluation using DM/CGGI functional bootstrapping [CJP21], (2) tree-based method over extracted digits of the input ciphertext [GBA21], and (3) circuit-bootstrapping method using bit decomposition and multiplexer tree evaluation [BBB$^+$23]. The best state-of-the-art results for larger arbitrary LUT were achieved using the circuit-bootstrapping method, and are

Table 1: State-of-the-art experimental results for evaluating arbitrary-function LUTs of 10 bits and higher in a single-threaded CPU setting (note that [WHS+24] used AVX-512 acceleration). Results marked with an asterisk $*$ were obtained in the same experimental setting. We highlight in bold the best amortized time for a given LUT size. #Slots refers to the vector size that the LUT is evaluated over.

Work	Method	#Slots	LUT size (bits)	Latency (s)	Amz. time (ms)
[WHS+24]	CGGI	1	12	0.1706	170.6
[LW23]	BFV	2^{15}	12	1,280	39.1
[CCP24]	CKKS	2^{17}	10	3,892.84	29.70
			12	6,721.37	51.28
[BKSS24]	CKKS	2^{12}	10	50.3	12.3
[AKP24]*	CKKS	2^{17}	10	243.95	**1.86**
			12	706.98	5.39
			14	2,479.2	18.92
			16	45,722	348.83
Our work*	CKKS	2^{16}	10	151.71	2.31
			12	167.27	**2.55**
			14	335.88	**5.13**
			16	480.80	**7.34**
			18	1,290.5	**19.69**
		2^{17}	20	8,447.0	**64.45**

presented in [WHS+24]. As seen from Table 1, this approach attains the smallest latency, but its main drawback is that it evaluates an LUT only for a single encrypted integer (rather than a vector of integers). As a result, its amortized time for 12-bit LUTs is two orders of magnitude higher than for our best method, i.e., CMT-FBT.

The first practically efficient vectorized functional bootstrapping approach was proposed in [LW23] (there were multiple prior theoretical works, e.g., [DMKMS24], which did not outperform the regular CGGI bootstrapping in practice). This method works with LWE ciphertexts (just like DM/CGGI), but uses BFV for the core bootstrapping operation, evaluating a polynomial over a finite field. Table 1 shows that the amortized time for 12-bit LUTs is one order of magnitude higher than CMT-FBT. Moreover, this method scales poorly for higher LUT sizes, doubling the polynomial degree for each bit of plaintext space (for 12-bit LUTs, the polynomial degree is already 786433, i.e., close to 2^{20}).

There are also several methods for LUT evaluation based on CKKS. We start with leveled approaches, i.e., with CKKS without bootstrapping. Cheon et al. [CCP24] proposed an approach for computing larger LUTs by using CKKS in the leveled mode while also decomposing the messages on a smaller basis. The input message is first decomposed into selector bits, which are then used to compute a homomorphic binary tree. However, since the tree is evaluated in the leveled mode, efficiency largely depends on the depth of the tree. As a result, the amortized time for their method scales much worse with LUT size than CMT-FBT (see Table 1). Chung et al. [CKKL24] proposed a multi-variate approach for larger LUT sizes which reduces the polynomial degree from P to $p\lceil\log_p P\rceil$. However, their solution requires matrix multiplications (plaintext instead of scalar), assumes an initial complex encoding (which does not support multiplications between ciphertexts) and does not perform bootstrapping. [CKKL24] reported a GPU runtime of 0.58 milliseconds for 12-bit LUTs, but it is hard to compare their result with the entries in Table 1 as GPU runtimes are typically 1-2 orders of magnitude smaller than single-threaded CPU runtimes.

Since 2024, functional bootstrapping algorithms have also been proposed based on the CKKS scheme [BCKS24, BKSS24, AKP24]. These methods use trigonometric (Hermite) interpolations, which are expressed as polynomials of degree P. However, the performance of such functional bootstrapping techniques restricts their use to input plaintext spaces of at most 14 bits in practice [AKP24] (as illustrated in Table 2). Alexandru et al. [AKP24] also expanded the direct LUT evaluation to larger LUT evaluation using the tree-based method over ciphertext digits, which was originally proposed by Guimarães et al. [GBA21]. Nevertheless, the tree-based method translates to evaluating sequential LUTs with encrypted coefficients, which becomes very expensive in the CKKS setting (as compared to DM/CGGI), making this option more costly than direct functional bootstrapping.

The homomorphic evaluation of functions over large plaintext spaces using a multiprecision approach, where CKKS functional bootstrapping is used as a subroutine, has been considered in [Kim25, BK25]. However, this approach cannot support arbitrary-function LUTs and works only for implementing basic positional or residue number arithmetic.

1.4 Paper organization

In Section 2 we provide some background on RLWE secret-key encryption, the CKKS scheme and functional bootstrapping. We (re-)introduce multi-value bootstrapping for CKKS and present our functional digit decomposition algorithm in Section 3. In Sections 4 and 5, we describe our new functional bootstrapping algorithms that target large LUTs. In Section 6, we compare both algorithms and present some optimizations in Section 7. Section 8 provides timings and comparison of our new methods with the functional bootstrapping algorithm proposed in [AKP24]. We conclude with future research directions in Section 9.

2 Preliminaries

2.1 Notation

Vectors are presented in bold or as $(x_1, x_1, \ldots, x_n)^\intercal$. Matrices with a rows and b columns are denoted as being elements of $\mathcal{M}_{a,b}$. We denote the inverse vectorization by $\text{vec}^{-1}_{a,b} : \mathcal{M}_{ab,1} \to \mathcal{M}_{a,b}$, which turns column vectors of ab elements in matrices with a rows and b columns. All logarithms are in base 2 unless explicitly mentioned. For all $x \in \mathbb{R}$, $\lfloor x \rfloor$, $\lfloor x \rceil$ and $\lceil x \rceil$ correspond to the rounding to the previous, closest, and next integer, respectively. For $a, b \in \mathbb{N}$ with $a < b$, $[\![a, b]\!]$ denotes the set of integers $\{a, a+1, ..., b\}$. For $i, j \in \mathbb{N}$, the Kronecker delta $\delta_{i,j}$ is defined as follows:

$$\delta_{i,j} := \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

Given the ring dimension N, chosen as a power of two for efficiency, let $\mathcal{R}$ be the ring of polynomials of integer coefficients defined by $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$. For an integer $q > 1$, $\mathcal{R}_q$ is the quotient ring defined by $\mathcal{R}_q = \mathcal{R}/q\mathcal{R} = \mathbb{Z}_q[X]/(X^N + 1)$. A polynomial $m(X) \in \mathcal{R}$ or $\mathcal{R}_q$ is represented by the vector of its coefficients $\mathbf{m}$. We will use the two notations interchangeably, e.g., $\mathbf{m} = m(x) \in \mathcal{R}_q$. For integers p and q, $[p]_q$ denotes the modular reduction of p modulo q. We refer to the modulo q function by mod_q. For a vector or a polynomial, modular reduction is applied component-wise.

We denote the sets of positive natural and real numbers as $\mathbb{N}^+$ and $\mathbb{R}^+$, respectively. For an abitrary function $f : \mathbb{Z}_p \to \mathbb{Z}_p$ with $p \in \mathbb{N}^+$, we associate a *p-to-p look-up table* $\mathsf{LUT}(f)$.

2.2 RLWE encryption

The secret-key RLWE encryption [LPR10] is parameterized by a plaintext modulus p, a ciphertext modulus q, a ring dimension N and two small distributions $\chi_{\mathsf{key}}, \chi_{\mathsf{err}}$ over $\mathcal{R}$. Encryption and decryption of a message $m(X) \in \mathcal{R}_p$ satisfy:

- $\mathsf{Encrypt}$: For a uniformly sampled $a \leftarrow \mathcal{R}_q$ and $\mathsf{sk} \leftarrow \chi_{\mathsf{key}}$, $e \leftarrow \chi_{\mathsf{err}}$, the encryption of $m(X)$ is defined as: $\mathsf{Encrypt}(m, \mathsf{sk}) = (-a \cdot \mathsf{sk} + (q/p) \cdot m + e, a) = (b, a) \in \mathcal{R}_q^2$.
- $\mathsf{Decrypt}$: Given a ciphertext $c = (b, a) \in \mathcal{R}_q^2$ with underlying plaintext $m(X) \in \mathcal{R}_p$, the decryption is given by: $\mathsf{Decrypt}((b, a), \mathsf{sk}) = \lfloor (p/q) \cdot (b + a \cdot \mathsf{sk}) \rceil$.

We also define the modulus switching operation:

- $\mathsf{ModSwitch}$: Given a ciphertext $c = (b, a) \in \mathcal{R}_q^2$ and a new modulus Q, modulus switching from q to Q is defined as: $\mathsf{ModSwitch}(c, q, Q) = (\lfloor Q/q \cdot b \rceil, \lfloor Q/q \cdot a \rceil) \in \mathcal{R}_Q^2$.

2.3 CKKS

In this work, we consider the RNS variant [CHK+18b, BGP+20] of the CKKS scheme [CKKS17]. To differentiate between the RLWE and CKKS parameters, we denote the CKKS ciphertext modulus with an apostrophe, for instance, q'_0 for the CKKS base modulus (as opposed to q for the RLWE ciphertext modulus).

We select $L + 1$ distinct prime numbers $q'_0, \ldots, q'_L$ and define for all $\ell \in [\![0, L]\!]$, $Q'_\ell = \prod_{i=0}^{\ell} q'_i$, where L is the maximum multiplicative depth. We represent elements of $\mathbb{Z}_{Q'_L}$, i.e., RNS limbs, by their Chinese Remainder Theorem (CRT) decomposition in $\mathbb{Z}_{q'_0} \times \cdots \times \mathbb{Z}_{q'_L}$.

For a given ring dimension N, we set $M = 2N$ and the scaling factor $\Delta = 2^\rho$ for $\rho \in \mathbb{N}^+$, with $\Delta < q'_0$. We use the distribution $\chi_{\mathsf{key}'}$ over $\mathcal{R}$ during key generation to sample N coefficients in $\{-1, 0, 1\}$ with exactly h non-zero coefficients. We also define an additional distribution χ_{pkenc} over $\mathcal{R}$ for key randomization before encryption.

A message is a vector $\mathbf{z}$ in $\mathbb{C}^n$ such that $n \mid N/2$. A plaintext pt is a polynomial in $\mathcal{R}$. We use the canonical embedding to switch between messages encoded as vectors and their corresponding (slot-encoded) plaintexts with the Encode and Decode procedures. We refer to the canonical embedding for a given n by $\tau'_n : \mathbb{R}[X]/(X^N + 1) \to \mathbb{C}^n$ where $\tau'_n(a) = (a(\zeta^j))_{j \in \mathbb{Z}_M^*}$, with $\zeta = \exp(2\pi i/M)$ and $\mathbb{Z}_M^* = \{x \in \mathbb{Z}_M \mid \mathsf{gcd}(x, M) = 1\}$. We denote a ciphertext at level $\ell \in [\![0, L]\!]$ by the tuple $\{\mathsf{ct}, Q'_\ell\}$ where $\mathsf{ct} \in \mathcal{R}^2_{Q'_\ell}$.

In the following, we review CKKS key generation and encryption-related algorithms. The homomorphic operations on CKKS ciphertexts are described in Appendix A.

- $\mathsf{Setup}(\lambda)$: For a given security parameter λ and the maximum level $L \in \mathbb{N}$, output the ring dimension N and set the small distributions $\chi_{\mathsf{key}'}$, χ_{err} and χ_{pkenc}.
- KeyGen: Sample an $a \leftarrow \mathcal{R}_{Q'_L}$, a secret $s \leftarrow \chi_{\mathsf{key}'}$ and an error $e \leftarrow \chi_{\mathsf{err}}$. Set $\mathsf{sk} = (1, s)$ and $\mathsf{pk} \leftarrow (b, a)$ with $b = [-a \cdot s + e]_{Q'_L}$.
- $\mathsf{Encode}(\mathbf{z})$: For $\mathbf{z} \in \mathbb{C}^n$, return $\mathsf{pt} \leftarrow \lfloor \tau_n'^{-1}(\Delta \cdot \mathbf{z}) \rceil$.
- $\mathsf{Decode}(\mathsf{pt})$: For $\mathsf{pt} \in \mathcal{R}$, return $\mathbf{z} \leftarrow \tau'_n(\mathsf{pt}/\Delta)$.
- $\mathsf{Encrypt}(\mathsf{pt}, \mathsf{pk})$: For $\mathsf{pt} \in \mathcal{R}$, sample $u \leftarrow \chi_{\mathsf{pkenc}}$ and $e_0, e_1 \leftarrow \chi_{\mathsf{err}}$, set $\mathsf{ct} \leftarrow [u \cdot \mathsf{pk} + (\mathsf{pt} + e_0, e_1)]_{Q'_L}$ and return $\{\mathsf{ct}, Q'_L\}$.
- $\mathsf{Decrypt}(\{\mathsf{ct}, Q'_\ell\}, \mathsf{sk})$: For $\mathsf{ct} = (c_0, c_1) \in \mathcal{R}^2_{Q'_\ell}$, return $\hat{m} = c_0 + c_1 \cdot s$.

2.4 CKKS functional bootstrapping

The original CKKS bootstrapping [CHK+18a] aimed at regaining multiplicative depth, enabling further computations on exhausted ciphertexts. These are ciphertexts that can no longer support multiplication, that is, with a ciphertext modulus reduced to q'_0. In this sense, we can consider CKKS bootstrapping as an analogue of CGGI's circuit bootstrapping, as both are multiplication enablers. Recent works [AKP24, BKSS24] introduced the notion of functional bootstrapping for CKKS, which allows the evaluation of arbitrary functions while bootstrapping the ciphertexts.

In this work, we focus on the procedure presented in [AKP24]. We start with an RLWE ciphertext that encodes integers from $\mathbb{Z}_p^n$ where p is a few bits long. We aim to obtain a bootstrapped RLWE ciphertext encrypting the evaluation of an arbitrary function $f : \mathbb{Z}_p \to \mathbb{Z}_p$ (or $\mathsf{LUT}(f)$) on these integers. We will rely on the first-order Hermite interpolation[2] of the function f and we refer to it as $T_f \in \mathbb{C}[X]$. T_f satisfies:

$$\mathrm{Re}(T_f(\exp(2i\pi k/p))) = f(k),\ \mathrm{Re}(T_f'(\exp(2i\pi k/p))) = 0, \quad \forall k \in [\![0, p-1]\!].$$

The evaluation of this polynomial interpolation allows both the reduction of noise (thanks to the first derivative set to 0) and the evaluation of the interpolated function. A more detailed explanation is given in [AKP24]. We use similar notations for ease of understanding.

We describe the functional bootstrapping procedure for an RLWE ciphertext:[3]

- ModSwitch: We start by converting the RLWE ciphertext ct to a corresponding (exhausted) CKKS ciphertext ct'. To do so, we switch the modulus from q, the RLWE ciphertext modulus, to q_0', the CKKS base modulus. Then, we adjust the scaling factor by Δ/q_0' to obtain a CKKS ciphertext encoding $\Delta \cdot m(X)/p$.

- ModRaise and CoeffsToSlots: Similarly to CKKS classical bootstrapping, we raise the modulus to Q_L' and run the encoding homomorphically. We obtain $t(X) = \Delta m(X)/p + q_0' I(X)$ after modulus raise, which contains $q_0'-$overflows denoted by I. After CoeffsToSlots, each coefficient t_i of $t(X)$ is stored in a different slot.

- EvalExp: We use a Chebyshev interpolation to approximate the complex exponential $\hat{m}_i \approx \exp(2\pi i m_i/p)$ for each t_i, which performs the modular reduction modulo p to remove the overflows.

- EvalLUT: We then compute the trigonometric Hermite interpolation T_f for a given $\mathsf{LUT}(f)$, and take the real part. This gives us $\tilde{m}_i \approx f(m_i)$.

- SlotsToCoeffs: Then, we put the result back in the coefficient encoding by homomorphically evaluating the decoding and apply a scaling by $Q'/(\Delta p)$, which gives us an encryption of $Q'\tilde{m}(X)/p$ with Q' being the modulus after bootstrapping.

- ModSwitch: We perform a final modulus switch from Q' to Q to get an RLWE ciphertext encoding $Q\tilde{m}(X)/p$, with ciphertext modulus Q.

We can perform additional leveled CKKS computations before switching back to an RLWE ciphertext. We will eventually need to adjust Q' to match the modulus after the bootstrapping and leveled computations.

We will later refer to this algorithm as AKP-FBT, which takes as input an RLWE ciphertext ct, an LUT, and returns an RLWE ciphertext corresponding to the evaluation of the LUT on all elements of the encrypted vector. This algorithm consumes more depth than classical bootstrapping, as we need to evaluate a degree $p-1$ polynomial (for the trigonometric Hermite interpolation) after the approximate modular reduction, but can support smaller scaling CKKS factors for smaller plaintext sizes.

[2]A higher-order Hermite interpolation could be used for further noise cleaning as suggested in [AKP24] at the cost of evaluating a polynomial of a higher degree.

[3]A similar procedure exists for CKKS ciphertexts with a different order of operations, namely SlotsToCoeffs→ ModRaise→ CoeffsToSlots→ EvalExp→ EvalLUT. It is described in more detail in [AKP24].

Bootstrapping failure probability. There are two sources of error that can lead to a functional bootstrapping failure. The first potential cause (same as in classical CKKS bootstrapping) is due to underestimating the interpolation interval when removing the overflow term after the modulus raising step. The probability of such a failure in our setting is comparable to that in typical CGGI bootstrapping (see Table 5.6 of [BCC+25] for common CGGI bootstrapping failure probabilities), but it can be made practically negligible, e.g., below $< 2^{-128}$, without increasing the parameters and affecting the efficiency using the sparse-secret encapsulation technique from [BTPH22] (this technique was recently implemented in OpenFHE v1.4.0 [AAB+22] and can be incorporated in our implementation). The second source of error is due to noise accumulation via homomorphic computations during functional bootstrapping. We focus on the latter in this paper and take advantage of noise reduction via Hermite interpolation to make the decryption rounding error practically negligible.

3 Building blocks: Multi-value bootstrapping and functional digit decomposition

Multi-value bootstrapping (MVB) and digit decomposition were presented as extensions to AKP-FBT in [AKP24]. We provide experimental results and a complexity estimation of MVB, and modify digit decomposition to support arbitrary input and output sizes, as well as function evaluations on the digits.

3.1 Multi-value bootstrapping

MVB was first introduced for DM/CGGI in [CIM19], and allows to evaluate multiple LUTs during a single bootstrapping operation, with small overhead. AKP-FBT also supports this functionality, through some changes in the EvalLUT step. Instead of directly computing the trigonometric Hermite interpolation associated with a single LUT, we first precompute a power tower $\{\exp(2\pi i x/p)^j\}_{j\in I_{\mathsf{PolyPS}}}$. We can then use these precomputed values to evaluate the (degree $p-1$) Hermite interpolation of each LUT using the Paterson-Stockmeyer algorithm [PS73] PolyPS, resulting in k ciphertexts for k functions. Finally, we perform SlotsToCoeffs on each of the k ciphertexts.

We describe MVB for an input ciphertext ct and a set of k LUTs $\{\mathsf{LUT}(f_i)\}_{1\le i\le k}$ in Algorithm 1. Similarly to AKP-FBT, Q denotes the output modulus for the RLWE ciphertext, and p represents both the input and output plaintext modulus.

Correctness of Algorithm 1. The correctness of MVB follows from the correctness of AKP-FBT (detailed in [AKP24]).

Complexity of Algorithm 1. The complexity of MVB can be deduced from the complexity of AKP-FBT. The same multiplicative depth is needed, and the complexity is identical until CoeffsToSlots. Then, we compute the powers of $E(x) = \exp(2\pi i x/p)$ (with EvalPower) for an evaluation of the polynomial interpolation of a LUT using PolyPS, which requires $\lceil\sqrt{2d}+\log d\rceil + O(1)$ non-scalar multiplications for a degree d polynomial, out of which the reusable powers computation requires $\lceil\sqrt{d/2}+\log d + O(1)\rceil$ non-scalar multiplications and the rest of the polynomial evaluation requires $\lceil\sqrt{d/2}+O(1)\rceil$. We then use those powers to evaluate k such interpolations, and perform k SlotsToCoeffs and ModSwitch to convert back to RLWE ciphertexts. Compared to AKP-FBT, we perform $(k-1)$ additional Paterson-Stockmeyer evaluations without the need to compute the powers of $E(x)$, and $(k-1)$ SlotsToCoeffs and ModSwitch.

Algorithm 1 Multi-value bootstrapping for an RLWE ciphertext

```
procedure MVB($\mathsf{ct} \in \mathcal{R}_q^2, \{\mathsf{LUT}(f_i)\}_{1\leq i\leq k}, p, k$)
 1: $\mathsf{ct}_1 \leftarrow \mathsf{ModSwitch}(\mathsf{ct}, q'_0)$
 2: $\mathsf{ct}_2 \leftarrow \mathsf{ModRaise}(\Delta \cdot \mathsf{ct}_1/q'_0, Q'_L)$
 3: $\mathsf{ct}_3 \leftarrow \mathsf{CoeffsToSlots}(\mathsf{ct}_2)$
 4: $\mathsf{ct}_{\text{exp}} \leftarrow \mathsf{EvalExp}(\mathsf{ct}_3)$        ▷ Approximation of $x \mapsto \exp(2\pi i x/p)$
 5: $\{\mathsf{ct}_{\text{pow},j}\}_{j\in I_{\mathsf{PolyPS}}} \leftarrow \mathsf{EvalPower}(\mathsf{ct}_{\text{exp}}, p)$
 6: for $i \in [\![1,k]\!]$ do
 7:     $\mathsf{ct}_{4,i} \leftarrow \mathsf{PolyPS}(\{\mathsf{ct}_{\text{pow},j}\}_{j\in I_{\mathsf{PolyPS}}}, \mathsf{LUT}(f_i))$ ▷ Evaluates the Hermite interpolation of $f_i$ using the precomputed powers and the Paterson-Stockmeyer algorithm
 8:     $\mathsf{ct}_{5,i} \leftarrow \mathsf{MulConst}(\mathsf{Add}(\mathsf{ct}_{4,i}, \mathsf{Conj}(\mathsf{ct}_{4,i}), 0.5)$        ▷ Computes the real part
 9: for $i \in [\![1,k]\!]$ do        ▷ Switch all of the results back to RLWE ciphertexts
10:     $\mathsf{ct}_{6,i} \leftarrow \mathsf{SlotsToCoeffs}(\mathsf{ct}_{5,i})$
11:     $\mathsf{ct}_{7,i} \leftarrow \mathsf{ModSwitch}(\mathsf{ct}_{6,i}, Q)$
12: return $\{\mathsf{ct}_{7,i}\}_{1\leq i\leq k}$
```

For a single-threaded execution, the cost of MVB can be estimated as:

$$\mathcal{C}_{\mathsf{MVB}} = \mathcal{C}_{\mathsf{AKP\text{-}FBT}} + (k-1)\cdot(\mathcal{C}_{\mathsf{PolyPS}} + \mathcal{C}_{\mathsf{StC}} + \mathcal{C}_{\mathsf{ModSwitch}}) < k \cdot \mathcal{C}_{\mathsf{AKP\text{-}FBT}}.$$

In this cost, only $\mathcal{C}_{\mathsf{AKP\text{-}FBT}}$ and $\mathcal{C}_{\mathsf{PolyPS}}$ depend on the size of the LUT. More precisely, in AKP-FBT, only the cost of EvalPower and PolyPS increase as p increases. Therefore, for large LUTs, the cost is dominated by $\mathcal{C}_{\mathsf{EvalPower}} + k \cdot \mathcal{C}_{\mathsf{PolyPS}}$. With multithreading, the k PolyPS, SlotsToCoeffs and ModSwitch can be done in parallel, meaning that $\mathcal{C}_{\mathsf{MVB}} = \mathcal{C}_{\mathsf{AKP\text{-}FBT}}$.

We provide in Table 2 a profiling of AKP-FBT, with the percentage of time taken by each step of the functional bootstrapping for LUTs of different sizes. We omit the final modulus switching as, in our case, we will perform further computations in CKKS after MVB and because it generally accounts for less than 1% of the runtime of AKP-FBT.

Table 2: Profiling of the isolated functional bootstrapping for a random p-to-p LUTs on a random encrypted input, single-threaded. RLWE/CKKS parameters are given in Table A1 in Appendix B. More details about the experimental setup are given in Section 8.

$\log p$	ModRaise + CtS	EvalExp	EvalPower	PolyPS	StC	Latency (s)
1	43.79%	44.15%	0.05%	0.01%	10.69%	16.69
2	36.55%	51.20%	2.49%	1.41%	7.03%	24.39
4	33.73%	49.20%	5.85%	2.62%	7.48%	30.12
6	30.27%	45.26%	9.68%	8.32%	5.62%	39.93
8	28.84%	32.61%	12.49%	21.20%	4.29%	56.59
10	12.34%	18.90%	15.01%	50.59%	2.83%	243.95
12	4.87%	7.66%	11.44%	74.79%	1.12%	706.98
14	1.49%	2.39%	7.02%	88.74%	0.33%	2479.2
16	0.47%	0.90%	4.80%	93.78%	0.04%	45722

We can notice that, as p increases, so does the percentage of time taken by PolyPS.[4]. Starting at 10 bits, it is already the predominant part of AKP-FBT. For 12 bits or more, PolyPS accounts for 75% or more of the runtime for AKP-FBT.

[4]The non-linear growth of the cost of PolyPS compared to EvalPower highlights the impact of the number of levels (increasing as p grows) on the cost of operations such as scalar-ciphertext multiplications, which are usually considered cheap, and the cost of the deep Paterson-Stockmeyer recursion, despite the similar complexity of the non-scalar multiplications.

We will use a slightly modified version of the MVB algorithm called SlotsMVB, which does not execute lines 10 to 12 to keep the values in the slot encoding. This will allow us to perform additional CKKS computations before switching back to RLWE.

3.2 Functional digit decomposition

The digit decomposition procedure for an RLWE ciphertext presented in [AKP24] relies on AKP-FBT and modulus switching. It is used to decompose an input in $\mathbb{Z}_P$ into digits in $\mathbb{Z}_p$, with $p < P$. The general idea is that for a value in $\mathbb{Z}_P$, we recursively extract the least significant digit in $\mathbb{Z}_p$, subtract it from the original value, and divide it by p.

We make several changes compared to the original algorithm in [AKP24]. We minimize the starting modulus and more efficiently handle the cases where $\log p > 1$ does not divide $\log P$, by more precisely adjusting P and Q. More significantly, we return CKKS ciphertexts instead of RLWE ciphertexts, and allow for the evaluation of one or more functions on each extracted digit using MVB.

To simplify the description of the functional digit decomposition algorithm, we consider the values p, P, q, and Q to be powers of two. Furthermore, we require the equality $p \cdot Q = q \cdot P$ to hold. We enforce the requirement by setting $\log Q = \log q + \log P - \log p$. We denote by $n_d = \lceil \log P / \log p \rceil$ the number of digits in $\mathbb{Z}_p$ needed to represent a number in $\mathbb{Z}_P$. Finally, we denote by $\{\mathsf{LUT}(f_i)\}_{1 \le i \le k}$ the LUTs for functions $f_i : \mathbb{Z}_p \to \mathbb{Z}_p$.

Algorithm 2 Functional digit decomposition for an RLWE ciphertext

```
procedure FDD($\mathsf{ct} \in \mathcal{R}_Q^2, \{\mathsf{LUT}(f_i)\}_{1 \le i \le k}, p, P$)
1:  $n_d \leftarrow \lceil \log P / \log p \rceil$
2:  for $i \in [\![1, n_d]\!]$ do
3:      $\{\mathsf{ctResult}\}_{0 \le j \le k} \leftarrow \mathsf{SlotsMVB}_p([\mathsf{ct}]_q, \{\mathsf{LUT}(\mathsf{mod}_p)\} \cup \{\mathsf{LUT}(f_j)\}_{1 \le j \le k})$  ▷ The first
    ciphertext encrypts the $\mathsf{mod}_p$ evaluation and the subsequent ciphertexts encrypt the
    $\{f_j\}_{1 \le j \le k}$ evaluations.
4:      $\mathsf{ct}_i \leftarrow \{\mathsf{ctResult}_j\}_{0 \le j \le k}$
5:      if $i < n_d$ then
6:          $\mathsf{ctMod} \leftarrow \mathsf{ModSwitch}(\mathsf{SlotsToCoeffs}(\mathsf{ctResult}_0), Q)$    ▷ Switch back to RLWE
7:          $\mathsf{ct} \leftarrow \mathsf{ct} - \mathsf{ctMod}$
8:          $\mathsf{ct} \leftarrow \mathsf{ModSwitch}(\mathsf{ct}, Q/p)$
9:          $P \leftarrow P / \min(p, P/p)$, $Q \leftarrow Q / \min(p, P/p)$  ▷ For all iterations except the last
    one, the digit size is $\log p$ bits; for the last iteration, the digit size is $\log(P/p)$ bits.
10: return $(\{\mathsf{ct}_i\}_{1 \le i \le n_d})$
```

Correctness of Algorithm 2. The correctness of FDD follows from the correctness of MVB and from the equality $p \cdot Q = q \cdot P$.

Complexity of Algorithm 2. The algorithm requires $\lceil \log_p P \rceil$ calls to SlotsMVB of width $k+1$ for k input LUTs, as well as $\lceil \log_p P \rceil - 1$ SlotsToCoeffs and subtractions, and $2\lceil \log_p P \rceil - 2$ ModSwitch. In the case we do not want to evaluate any additional function, FDD will only return the digit decomposition, and the $\lceil \log_p P \rceil$ SlotsMVB are equivalent to $\lceil \log_p P \rceil$ AKP-FBT without the return to RLWE.

4 Binary multiplexer tree method

Our initial approach to build a procedure for efficient homomorphic evaluation of LUTs is inspired by the multiplexer tree approach used in the CGGI setting [CGGI20, BBB+23].

The high-level idea in the CGGI case is to decompose the input LWE ciphertext into ciphertexts for each bit of the message, invoke (expensive) circuit bootstrapping to enable RGSW multiplications, and then evaluate a multiplexer tree using many homomorphic multiplications. In contrast, the AKP functional bootstrapping cheaply switches between RLWE and CKKS, and relies on CKKS for homomorphic multiplications instead of RGSW, which results in somewhat different constraints for building and optimizing our procedure. For instance, the cost of "circuit bootstrapping" in CKKS is relatively small (as compared to CGGI) but the relative cost of multiplications is higher (as we deal with evaluating ciphertexts at different levels instead of using the asymmetric noise growth available in RGSW). In this section, we will build the binary tree multiplexer procedure for AKP-FBT, showing step by step why we make specific algorithmic choices.

Let P be a power of two, which corresponds to the size of $\mathsf{LUT}(f)$ that we want to evaluate. We will denote by $\mathbf{y} = (y_0, \ldots, y_{P-1})^\intercal \in \mathcal{M}_{P,1}(\mathbb{Z}_P)$ the vector corresponding to $y_i = f(i)$, for $i \in [\![0, P-1]\!]$.

The function $f(x)$ can be evaluated with additions and multiplications using the Kronecker delta function:

$$f(x) = \sum_{i=0}^{P-1} \delta_{x,i} y_i. \tag{1}$$

Equation (1) can be rewritten using the binary decomposition of $x = (x_1, \ldots, x_{\log P})$ and index $i = (i_1, \ldots, i_{\log P})$ as

$$\sum_{i=0}^{P-1} \delta_{x,i} y_i = \sum_{i=0}^{P-1} y_i \prod_{j=1}^{\log P} \delta_{x_j, i_j}. \tag{2}$$

Noticing that for all $i \in [\![1, \log P]\!]$, $\delta_{x_i,0} = (1 - x_i)$ and $\delta_{x_i,1} = x_i$, we can compute expression (2) from the binary decomposition of x using only additions and multiplications. This method to compute an LUT is often referred to as the CMux tree method and is widely used in the CGGI setting [BBB+23]. We next explain how to efficiently adapt this paradigm of LUT computation to the CKKS setting.

4.1 Efficient evaluation of a binary multiplexer tree in CKKS

Out of the many different ways to evaluate Equation (2), we will focus on the one that reduces the complexity of evaluation in CKKS by applying the following guidelines:

- Minimize the multiplicative depth to reduce the number of RNS limbs (and potentially the ring dimension), resulting in improved computational complexity. The effect of reducing the multiplicative depth becomes more pronounced for larger LUTs.
- Minimize the number of ciphertext-ciphertext multiplications as these are more expensive than scalar ciphertext multiplications; this optimization has a more significant effect for smaller LUTs.
- Minimize the cost of ciphertext additions and scalar ciphertext multiplications (by evaluating them at the lowest possible number of RNS limbs). These operations become the bottleneck for larger LUTs.

To implement these guidelines, we rewrite Equation (2) in matrix form using Kronecker products, which we denote as $\otimes$. The goal is to put the LUT values in a matrix, and to select the row and column according to the corresponding digit. For $\alpha = 2^{\lceil \log \sqrt{P} \rceil}$ and $\beta = 2^{\log P - \lceil \log \sqrt{P} \rceil}$, we obtain

$$f(x) = \left(\bigotimes_{i=1}^{\lceil \log \sqrt{P} \rceil} \begin{pmatrix} 1 - x_i \\ x_i \end{pmatrix} \right)^\intercal \mathrm{vec}_{\alpha,\beta}^{-1}(\mathbf{y}) \left(\bigotimes_{i=\lceil \log \sqrt{P} \rceil + 1}^{\log P} \begin{pmatrix} 1 - x_i \\ x_i \end{pmatrix} \right). \tag{3}$$

In Equation (3), the Kronecker products are evaluated in the order of increasing indices, and $\binom{1-x_i}{x_i}$ are column vectors with two components.

When $\log P$ is a power of two, both the Kronecker products on the left and on the right output vectors of size $\sqrt{P} = \alpha = \beta$, and the matrix products enable selecting the correct row and column in the square matrix $\text{vec}^{-1}_{\sqrt{P},\sqrt{P}}(\mathbf{y})$ to yield the result.

For example, for $P = 16$, the LSB part of the computation involves the product for the first two bits, x_1 and x_2, as follows:

$$\mathbf{v}_{\mathsf{LSB}} = \begin{pmatrix} 1-x_1 \\ x_1 \end{pmatrix} \otimes \begin{pmatrix} 1-x_2 \\ x_2 \end{pmatrix} = \begin{pmatrix} (1-x_1)\cdot(1-x_2) \\ (1-x_1)\cdot x_2 \\ x_1 \cdot (1-x_2) \\ x_1 \cdot x_2 \end{pmatrix} = \begin{pmatrix} \delta_{x_1,0}\cdot\delta_{x_2,0} \\ \delta_{x_1,0}\cdot\delta_{x_2,1} \\ \delta_{x_1,1}\cdot\delta_{x_2,0} \\ \delta_{x_1,1}\cdot\delta_{x_2,1} \end{pmatrix} = \begin{pmatrix} \delta_{2x_1+x_2,0} \\ \delta_{2x_1+x_2,1} \\ \delta_{2x_1+x_2,2} \\ \delta_{2x_1+x_2,3} \end{pmatrix}$$

This resulting column vector of size $\sqrt{P} = 4$ contains encryptions of the four possible products of selector bits, namely, three zeros and a single one.

For our CKKS implementation of the binary multiplexer tree (further described below in Algorithm 4), we first perform the two Kronecker products using the binary tree method to minimize the multiplicative depth. Then, we multiply the inverse vectorization of $\mathbf{y}$ by the second Kronecker product. Finally, we multiply the result by the first Kronecker product. This evaluation order allows one to minimize the number of RNS limbs when multiplying by the inverse vectorization, i.e., to minimize the complexity of computing ciphertext additions and scalar ciphertext multiplications. Note that the choice of $\sqrt{P}$ minimizes the number of ciphertext-ciphertext multiplications. In summary, this order of evaluation and this choice of parameters allow one to implement all three optimization guidelines in the same procedure.

To evaluate the Kronecker products, we use Algorithm 3, which takes a vector of vectors of ciphertexts as input, and performs the products in a binary tree manner to reduce multiplicative depth. For generality (as we will use this procedure later in the paper), the algorithm can take as input $n \in \mathbb{N}^+$ vectors of ciphertexts, each of length $p_i \in \mathbb{N}^+$.

Algorithm 3 Kronecker products of CKKS ciphertext vectors

```
procedure KProduct($\{\mathsf{ctVec}_r\}_{1\le r\le n} \in (R^2_{Q'_\ell})^{p_1} \times \cdots \times (R^2_{Q'_\ell})^{p_n}$)
1:  for $i \in [\![1, \lceil \log n \rceil]\!]$ do
2:      for $j \in [\![1, \lfloor n/2 \rfloor]\!]$ do
3:          $l_1 = \mathtt{length}(\mathsf{ctVec}_{2j-1})$, $l_2 = \mathtt{length}(\mathsf{ctVec}_{2j})$
4:          $\mathsf{ctVec}_j = \{\mathsf{Mul}(\mathsf{ctVec}_{2j-1}[((k-1) \mod l_1)+1], \mathsf{ctVec}_{2j}[\lfloor (k-1)/l_1 \rfloor + 1])\}_{k=1}^{l_1 l_2}$
5:      if $n$ mod $2 = 1$ then
6:          $\mathsf{ctVec}_{\lceil n/2 \rceil} = \mathsf{ctVec}_n$
7:      $n \leftarrow \lceil n/2 \rceil$
    return $\mathsf{ctVec}_1 \in (R^2_{Q'_{\ell-\lceil \log n \rceil}})^{p_1 \times \cdots \times p_n}$
```

Correctness and noise analysis of Algorithm 3. The correctness of KProduct follows from the correctness of CKKS multiplication and the associativity of the Kronecker product. For input ciphertexts with noise bounded by $\varepsilon_{\mathsf{in}}$, the KProduct algorithm results in moderate noise growth, as it only performs ciphertext multiplications of bits. In the worst case, we multiply only ones together. Because the message is bounded by one at each step, the noise accumulation is additive rather than multiplicative. With $B_{\mathsf{err}} \in \mathbb{R}^+$ the upper bound of the noise introduced by relinearization and rescaling, the worst-case output noise of the multiplication of two ciphertexts encrypting binary messages with input noise $\varepsilon_{\mathsf{in}}$ is

$$2|\varepsilon_{\mathsf{in}}| + |\varepsilon_{\mathsf{in}}|^2/\Delta + B_{\mathsf{err}}.$$

This allows us to bound the output noise by

$$|\varepsilon_{\mathsf{out}}| \leq n \cdot (|\varepsilon_{\mathsf{in}}| + B_{\mathsf{err}}) \,, \qquad \text{for } |\varepsilon_{\mathsf{in}}|^2/\Delta \leq B_{\mathsf{err}}.$$

The noise growth is, therefore, linear in the number of inputs n. Since the number of inputs to KProduct is logarithmic in the size of the LUT, the overall noise growth is logarithmic with respect to the LUT size.

Complexity of Algorithm 3. For n input vectors, the multiplicative depth is $\lceil \log n \rceil$. For simplicity, we assume that $p_1 = \cdots = p_n = p$ (which is a common case) and that n is a power of two. At each iteration of i in Algorithm 3, we perform $np^{2^i}/2^i$ ciphertext multiplications. That is, the total number of ciphertext multiplications is

$$\sum_{i=1}^{\log n} n\frac{p^{2^i}}{2^i} = n\left(\frac{p^2}{2} + \frac{p^4}{4} \cdots + \frac{p^n}{n}\right).$$

This sum is heavily dominated by its largest term, p^n. Therefore, the total number of multiplications is $\Theta(p^n)$. More precisely, the total number of ciphertext multiplications for $n \geq 2$ can be bounded within $[p^n, p^n + 4p^{n/2}]$. It is worth noting that the final step of the KProduct algorithm which performs exactly p^n multiplications (accounting for the vast majority of the multiplications) is executed using the minimum possible number of RNS limbs, and is therefore as inexpensive as possible.

Based on Equation (3) and our discussion above, we now formulate the binary multiplexer tree (BMT) algorithm to homomorphically evaluate an LUT, given the binary decomposition of a ciphertext. In Algorithm 4, the $\mathsf{ctDigit}_i$'s correspond to x_i's and $\mathsf{LUT}(f)$ to $\mathbf{y}$ in Equation (3).

Algorithm 4 Binary multiplexer tree evaluation for CKKS ciphertexts

```
procedure BMT($\{\mathsf{ctDigit}_i\}_{1\leq i\leq \log P} \in \mathcal{R}^2_{Q'_\ell}$, $\mathsf{LUT}(f)$)
1:  for $i \in [\![1, \log P]\!]$ do
2:      $\mathsf{ctDigitVec}_i \leftarrow \{\mathsf{AddConst}(-\mathsf{ctDigit}_i, 1), \mathsf{ctDigit}_i\}$ ▷ Instead of only storing the digit
    $x_i$, we create the vector $(1 - x_i, x_i)$

3:  $\mathsf{ctLSB} \leftarrow \{\mathsf{ctDigitVec}_i\}_{1\leq i\leq \lceil \log \sqrt{P}\rceil}$
4:  $\mathsf{ctMSB} \leftarrow \{\mathsf{ctDigitVec}_i\}_{\lceil \log \sqrt{P}\rceil < i\leq \log P}$

5:  $\mathsf{ctKLSB} \leftarrow \mathsf{KProduct}(\mathsf{ctLSB})$
6:  $\mathsf{ctKMSB} \leftarrow \mathsf{KProduct}(\mathsf{ctMSB})$

7:  $l_{\mathsf{LSB}} = \mathtt{length}(\mathsf{ctKLSB})$
8:  for $i \in [\![1, l_{\mathsf{LSB}}]\!]$ do
9:      $\mathsf{ctKMSBLut}[i] \leftarrow \sum_{j=1}^{\mathtt{length(ctKMSB)}} \mathsf{MulInt}\big(\mathsf{ctKMSB}[j], \mathsf{LUT}(f)[j + (i-1)\cdot l_{\mathsf{LSB}}]\big)$
10:     $\mathsf{ctResult}[i] = \mathsf{Mul}(\mathsf{ctKLSB}[i], \mathsf{ctKMSBLut}[i])$
    return $\sum_{i=1}^{l_{\mathsf{LSB}}} \mathsf{ctResult}[i]$
```

Complexity of Algorithm 4. We start by performing $\log P$ AddConst. Then, we perform two calls to KProduct, requiring $\lceil \log(\log P)\rceil - 1$ multiplicative depth and $\Theta(\sqrt{P})$ ciphertext multiplications. We then perform P scalar multiplications, $O(P)$ additions and $\sqrt{P}$ ciphertext multiplications. The total multiplicative depth is $\lceil \log(\log P)\rceil$, and most of the computations are done with the minimum number of RNS limbs.

Correctness and noise analysis of Algorithm 4. The correctness of BMT follows from the correctness of Equation (3), KProduct and CKKS operations. Let us assume that $\log\sqrt{P}$ is a power of two for simplicity. For a ciphertext with input noise bounded by $\varepsilon_{\mathsf{in}}$, there are three stages that contribute to the noise growth in Algorithm 4.

1. The KProduct computations on LSB and MSB sets involve multiplications of ciphertexts encrypting binary values, where noise accumulation is moderate and grows linearly with the number of input bits, i.e., $O(\log P)$.
2. The scalar multiplications (MulInt) in line 9 amplify the noise of the ctKMSB ciphertexts by the LUT coefficients, which can have a magnitude up to $P/2$. By then summing $\sqrt{P}$ ciphertexts, we further increase the noise by $\sqrt{P}$ in the worst case. This is the dominant source of noise amplification.
3. The final ciphertext multiplications by binary ciphertexts, and $\sqrt{P}$ ciphertexts summations in the last two lines introduce further noise. As ctKLSB encrypt binary values, the noise contribution is mainly that of the sum of $\sqrt{P}$ ciphertexts, which is $\sqrt{P}$ in the worst case.

The combination of these effects yields the overall worst-case noise bound of:

$$|\varepsilon_{\mathsf{out}}| = O\big(\underbrace{\sqrt{P}}_{(3)}\cdot\underbrace{\sqrt{P}\cdot P}_{(2)}\cdot\underbrace{\log P}_{(1)}\big)|\varepsilon_{\mathsf{in}}| = O(P^2\log P)|\varepsilon_{\mathsf{in}}|.$$

With the same reasoning and assuming that both summations over $\sqrt{P}$ ciphertexts deal with uncorrelated noise inputs, the average-case output noise bound can be estimated as

$$|\varepsilon_{\mathsf{out}}| = \Theta\big(P^{3/2}\log P\big)\cdot|\varepsilon_{\mathsf{in}}|.$$

For the result to be correct, we have to make sure that the inputs $\mathsf{ctDigit}_i$ have a small enough input error to guarantee correct decryption. This can be ensured for a large enough scaling factor by using the noise-cleaning techniques presented below.

4.2 Binary multiplexer tree functional bootstrapping

Using the modified ("non-functional" in this case) digit decomposition algorithm introduced in Section 3.2, we construct a new functional bootstrapping algorithm based on the BMT procedure, which we call BMT-FBT. However, we need an extra step between the functional digit decomposition and the binary multiplexer tree procedures to guarantee correctness for arbitrary input and digit sizes.

Noise cleaning. The digit decomposition uses AKP-FBT, which provides a first-order Hermite interpolation cleaning capability. Nevertheless, for large LUTs (especially in the case where the large plaintext size is decomposed in small digit sizes), this noise reduction is not sufficient. One solution is to use higher-order Hermite interpolation in AKP-FBT, but this turns out to be costly for $p = 2$ (see the first two rows in Table 1 [AKP24]). Therefore, we prefer the following less expensive alternative.

Concretely, we use the first-order polynomial Hermite interpolation proposed in BLEACH [DMPS24] (where it was treated as an ad-hoc cleaning polynomial) of the binary identity gate $G_{\mathsf{id}} = 3X^2 - 2X^3$. This polynomial has a horizontal tangent at 0 and 1, so that when writing its Taylor expansion for an input error $\varepsilon_{\mathsf{in}}$ we get $G_{\mathsf{id}}(x_i + \varepsilon_{\mathsf{in}}) = x_i + O(\varepsilon_{\mathsf{in}}^2)$, with $\varepsilon_{\mathsf{in}}^2 \leq |\varepsilon_{\mathsf{in}}|$. We only need two additional multiplicative levels to perform this cleaning on the $\log P$ binary digits $\mathsf{ctDigit}_i$, prior to the BMT computation. In practice, G_{id} is applied to the ciphertexts $\{\mathsf{ctDigit}_i\}$, which encrypt the binary components of the input message, to quadratically reduce their noise before they are input to the multiplexer tree.

To further reduce the magnitude of the error before the scalar multiplications, we can replace the ciphertext multiplications used in KProduct with a bivariate Hermite interpolation of the and gate, which corresponds to the product in the binary case. The polynomial $G_{\mathsf{and}} = 5X^2Y^2 - 2X^3Y^2 - 2X^2Y^3$ is defined so that it satisfies the truth table of the and gate: $G_{\mathsf{and}}(0,0) = G_{\mathsf{and}}(0,1) = G_{\mathsf{and}}(1,0) = 0$ and $G_{\mathsf{and}}(1,1) = 1$. The partial derivatives of G_{and} at $(0,0)$, $(0,1)$, $(1,0)$ and $(1,1)$ are set to 0 for error reduction. We need three multiplicative levels to perform multiplications using this polynomial instead of one.

Our new BMT-FBT procedure is described in Algorithm 5. First, it performs digit decomposition and performs cleaning for large input sizes. Then, the algorithm calls the BMT procedure on the encrypted bits. Finally, the result is converted back to the coefficient encoding and switched to the desired output modulus Q.

Algorithm 5 Binary multiplexer tree functional bootstrapping for an RLWE ciphertext

```
procedure BMT-FBT($\mathsf{ct} \in \mathcal{R}_Q^2, \mathsf{LUT}(f), P$)
1: $\{\mathsf{ctDigit}_i\}_{1\le i\le \log P} \leftarrow \mathsf{FDD}(\mathsf{ct}, \emptyset, 2, P)$  ▷ We do not pass additional LUTs to evaluate,
   and only get encrypted bits
2: $\mathsf{clean}(\{\mathsf{ctDigit}_i\}_{1\le i\le \log P})$  ▷ For large $P$, clean the binary digits
3: $\mathsf{ctResult}' \leftarrow \mathsf{BMT}(\{\mathsf{ctDigit}_i\}_{1\le i\le \log P}, \mathsf{LUT}(f))$  ▷ Algorithm 4
4: $\mathsf{ctResult} \leftarrow \mathsf{SlotsToCoeffs}(\mathsf{ctResult}')$  ▷ Switch back to the coefficient encoding
5: return $\mathsf{ModSwitch}(\mathsf{ctResult}, Q)$
```

Correctness and noise analysis of Algorithm 5. The correctness and complexity analysis of BMT-FBT follow from the correctness and complexity analysis of its building blocks. However, note that the output of BMT-FBT procedure should be an integer mod P in the RLWE coefficient encoding, therefore the result is correct if $|\varepsilon_{\mathsf{out}}| < \frac{1}{2P}$, where $\varepsilon_{\mathsf{out}}$ is the output ciphertext error. Without noise cleaning, the total noise accumulated as part of evaluating the complete BMT-FBT algorithm can exceed this bound, especially for large P, as the worst-case output noise of BMT is proportional to $P^2 \log P$. By using noise cleaning for $\log P > 12$ and setting a large enough scaling factor, we are able to absorb this noise growth and obtain correct results.

Complexity of Algorithm 5. The total multiplicative depth of the leveled computations is $\lceil \log(\log P) \rceil + \ell_{\mathsf{SlotsToCoeffs}}$, plus two levels if cleaning is performed. This comes from the two main parts in BMT-FBT:

- Precomputation with digit decomposition: the complexity is detailed in Section 3.2. We need $\log P$ calls to SlotsMVB on plaintext space $\mathbb{Z}_2$ with width 1, as well as negligible RLWE scheme operations like subtraction and modulus switching. Then for large P, we need to perform cleaning, i.e. evaluate the cleaning polynomial G_{id} on all the $\log P$ binary digits to reduce noise.
- LUT evaluation with BMT: the algorithm requires $\lceil \log(\log P) \rceil$ multiplicative depth, and performs $O(\sqrt{P})$ ciphertext multiplications and $O(P)$ integer multiplications and additions. We conclude by performing one SlotsToCoeffs and one ModSwitch to get back an RLWE ciphertext.

5 Collapsed multiplexer tree method

To further reduce the computational complexity of arbitrary-function LUT evaluation, we introduce a collapsed multiplexer tree method. Instead of working with a binary tree, we use a larger base p (assumed in this section to be a power of two for simplicity). This reduces the depth of the tree, and hence the multiplicative depth from $\lceil \log(\log P) \rceil$ to $\lceil \log \lceil \log_p P \rceil \rceil$ (excluding the extra noise cleaning from this analysis), but brings about other challenges.

5.1 Efficient evaluation of a general multiplexer tree in CKKS

We rewrite Equation (2) using the base p decomposition of x and index i:

$$\sum_{i=0}^{P-1} \delta_{x,i} y_i = \sum_{i=0}^{P-1} y_i \prod_{j=1}^{\lceil \log_p P \rceil} \delta_{x_j,i_j}. \tag{4}$$

Similarly, we rewrite Equation (3) for an arbitrary p, $\alpha = p^{\lceil \log_p \sqrt{P} \rceil}$, and $\beta = p^{\log_p P - \lceil \log_p \sqrt{P} \rceil}$ as

$$f(x) = \left(\bigotimes_{i=1}^{\lceil \log_p \sqrt{P} \rceil} \begin{pmatrix} \delta_{x_i,0} \\ \vdots \\ \delta_{x_i,p-1} \end{pmatrix} \right)^{\intercal} \mathrm{vec}_{\alpha,\beta}^{-1}(\mathbf{y}) \left(\bigotimes_{i=\lceil \log_p \sqrt{P} \rceil + 1}^{\lceil \log_p P \rceil} \begin{pmatrix} \delta_{x_i,0} \\ \vdots \\ \delta_{x_i,p-1} \end{pmatrix} \right). \tag{5}$$

This allows us to introduce the collapsed multiplexer tree (CMT) procedure (Algorithm 6), which is similar in structure to BMT, but has a key difference. Instead of taking as input only the binary digits of x, it takes the corresponding Kronecker delta functions. Note that at $p = 2$, CMT degenerates to BMT.

Algorithm 6 Collapsed multiplexer tree evaluation for CKKS ciphertexts

```
procedure CMT($\{\mathsf{ctKdelta}_i\}_{1 \le i \le \lceil \log_p P \rceil} \in (\mathcal{R}^2_{Q'_\ell})^p, \mathsf{LUT}(f)$)
1:  $\mathsf{ctLSD} \leftarrow \{\mathsf{ctKdelta}_i\}_{1 \le i \le \lceil \log_p \sqrt{P} \rceil}$
2:  $\mathsf{ctMSD} \leftarrow \{\mathsf{ctKdelta}_i\}_{\lceil \log_p \sqrt{P} \rceil < i \le \lceil \log_p P \rceil}$

3:  $\mathsf{ctKLSD} \leftarrow \mathsf{KProduct}(\mathsf{ctLSD})$
4:  $\mathsf{ctKMSD} \leftarrow \mathsf{KProduct}(\mathsf{ctMSD})$

5:  $l_{\mathsf{LSD}} = \mathtt{length}(\mathsf{ctKLSD})$
6:  for $i \in [\![1, l_{\mathsf{LSD}}]\!]$ do
7:      $\mathsf{ctKMSDLut}[i] \leftarrow \sum_{j=1}^{\mathtt{length}(\mathsf{ctKMSD})} \mathsf{MulInt}\big(\mathsf{ctKMSD}[j], \mathsf{LUT}(f)[j + (i-1) \cdot l_{\mathsf{LSD}}]\big)$
8:      $\mathsf{ctResult}[i] = \mathsf{Mul}(\mathsf{ctKLSD}[i], \mathsf{ctKMSDLUT}[i])$
    return $\sum_{i=1}^{l_{\mathsf{LSD}}} \mathsf{ctResult}[i]$
```

Correctness, complexity and noise analysis of Algorithm 6. The correctness, complexity and noise growth of CMT can be deduced in the same way as for Algorithm 4. More precisely, the correctness of the CMT procedure is a direct consequence of the algebraic identity in Equation (5), and the correctness of KProduct and CKKS. CMT consumes $\lceil \log \lceil \log_p P \rceil \rceil$ levels, as compared to $\lceil \log(\log P) \rceil$ for BMT. Similarly to BMT, CMT performs P MulInt, $O(\sqrt{P})$ additions and $\Theta(\sqrt{P})$ ciphertext multiplications. The worst-case (respectively average-case) output noise bound is therefore slightly smaller

$$|\varepsilon_{\mathsf{out}}| = O\big(P^2 \lceil \log_p P \rceil\big) \cdot |\varepsilon_{\mathsf{in}}| \qquad \Big(|\varepsilon_{\mathsf{out}}| = \Theta\big(P^{3/2} \lceil \log_p P \rceil\big) \cdot |\varepsilon_{\mathsf{in}}| \Big).$$

For p chosen to be small (a few bits), this bound is close to the one in the BMT algorithm.

5.2 Collapsed multiplexer tree functional bootstrapping

We can no longer rely on the formula of the binary case to compute the Kronecker delta functions. Instead, we represent the Kronecker delta as a p-to-2 LUT. The $\{\delta_{j,x_i}\}_{j=0}^{p-1}$ can be computed with p instances of functional bootstrapping. Importantly, we compute these p instances of functional bootstrapping in a single functional digit decomposition.

The resulting procedure, called CMT-FBT is described in Algorithm 7.

Algorithm 7 CMT-FBT for an RLWE ciphertext

```
procedure CMT-FBT($\mathsf{ct} \in \mathcal{R}_Q^2, \mathsf{LUT}(f), p, P$)
1: $\{\mathsf{ctFDD}_k\}_{1\leq k\leq \lceil \log_p P\rceil} \leftarrow \mathsf{FDD}(\mathsf{ct}, \{\mathsf{LUT}(\delta_k^p)\}_{0\leq k<p}, p, P)$    ▷ Algorithm 2
2: for $i \in [\![1, \lceil \log_p P\rceil]\!]$ do
3:     $\mathsf{ctKdelta}_i \leftarrow \mathsf{ctFDD}_i[1:p]$    ▷ Omit the first element, which is the digit
4: $\mathsf{clean}(\{\mathsf{ctKdelta}_i\}_{1\leq i\leq \lceil \log_p P\rceil})$    ▷ For large $P$, clean the Kronecker delta functions
5: $\mathsf{ctResult}' \leftarrow \mathsf{CMT}(\{\mathsf{ctKdelta}_i\}_{1\leq i\leq \lceil \log_p P\rceil}, \mathsf{LUT}(f))$    ▷ Algorithm 6
6: $\mathsf{ctResult} \leftarrow \mathsf{SlotsToCoeffs}(\mathsf{ctResult}')$    ▷ Switch back to the coefficient encoding
7: return $\mathsf{ModSwitch}(\mathsf{ctResult}, Q)$
```

Correctness and noise analysis of Algorithm 7. The correctness of CMT-FBT relies on its building blocks. As in BMT-FBT, the noise growth is primarily due to scalar multiplications and additions in CMT, leading to an output worst-case noise proportional to $P^2\lceil \log_p P\rceil$. The same condition on the output error must hold for correct decryption, that is $|\varepsilon_{\mathsf{out}}| < \frac{1}{2P}$. A formal and detailed noise analysis can be found in Appendix D.

The previously discussed noise reduction method, G_{id}, is used for the Kronecker delta binary ciphertexts $\{\mathsf{ctKdelta}_i\}$ for $\log P > 12$. Note that this is the same cleaning polynomial as used in the BMT-FBT case. Similarly, the G_{and} polynomial could also replace standard multiplications in the KProduct algorithm.

Complexity of Algorithm 7. The total multiplicative depth of the leveled computations is $\lceil \log\lceil \log_p P\rceil\rceil + \ell_{\mathsf{SlotsToCoeffs}}$, plus two levels if cleaning is performed. This comes from the two main parts in CMT-FBT:

- Precomputation with functional digit decomposition: the complexity is detailed in Section 3.2. We need $\lceil \log_p P\rceil$ calls to SlotsMVB on plaintext space $\mathbb{Z}_p$ with width $p+1$, as well as negligible RLWE scheme operations like subtraction and modulus switching. Then for large P, we need to perform cleaning, i.e., evaluate the cleaning polynomial G_{id} on all the $p \cdot \lceil \log_p P\rceil$ binary Kronecker deltas.
- LUT evaluation with CMT: the algorithm requires the $\lceil \log\lceil \log_p P\rceil\rceil$ multiplicative depth, and performs $O(\sqrt{P})$ ciphertext multiplications and $O(P)$ integer multiplications and additions. We conclude by performing one SlotsToCoeffs and one ModSwitch to get back an RLWE ciphertext.

Finally, we mention that several natural extensions of CMT-FBT (that also apply to BMT-FBT) are discussed in Appendix C, such as slot-independent LUT evaluation and multi-value CMT-FBT.

6 Comparison of BMT-FBT and CMT-FBT

In this section, we compare the costs of our multiplexer tree algorithms, BMT-FBT and CMT-FBT, and identify the range of LUT sizes for which each is more efficient. Note that reducing the multiplicative depth of the multiplexer tree from binary digits to larger digits introduces a need for MVBs to enable efficient functional digit decomposition. A priori, it is not clear whether this trade-off improves the total runtime, so we need to analyze each component separately.

First, we compare the theoretical cost of the precomputations, that is, the cost of the FDD in the binary and collapsed cases. We show that the complexity of the functional decomposition step based on MVB in CMT-FBT is always smaller than for BMT-FBT for $2 \leq \log p \leq 5$ in the single-threaded case. We then compare the cost of the BMT and CMT algorithms, and show that for a number of digits $\lceil \log_p P \rceil$ that is a power of two, the CMT algorithm will always outperform the BMT algorithm in the single-threaded case. Moreover, in the case of multi-threaded execution, the complexity of FDD in CMT-FBT is always smaller than for BMT-FBT for $p > 2$, and CMT always outperforms BMT.

In summary, the question of whether the trade-off offered by CMT-FBT can be advantageous is answered affirmatively. For a small decomposition basis and a power of two number of digits, we show that CMT-FBT always outperforms BMT-FBT.

6.1 Comparison of FDD for BMT-FBT and CMT-FBT

Let us first focus on the cost of FDD. As the full theoretical cost of AKP-FBT is rather complex, we will use the experimental results presented in Table 2 and Table A1 to model the runtime of FDD.

Single-threaded case. For a small p (less than 9 bits), the runtime of AKP-FBT is linear in the number of bits of the plaintext space $\log p$, so we can model it by $\mathcal{C}_{\mathsf{AKP\text{-}FBT}} : \log p \mapsto \alpha \log p + \beta$ for some $\alpha, \beta \in \mathbb{R}^+$. Additionally, we introduce $\gamma \in \mathbb{R}^+$, which models the theoretically constant runtime of SlotsToCoeffs and ModSwitch, and $\mathcal{C}_{\mathsf{PolyPS}}(p-1)$, which models the runtime of the evaluation of a degree $p-1$ polynomial with precomputed powers. This allows us to define the runtime of SlotsMVB of width $p+1$ on plaintext space $\mathbb{Z}_p$ as $\mathcal{C}_{\mathsf{SlotsMVB}} : \log p \mapsto \alpha \log p + \beta - \gamma + p \cdot \mathcal{C}_{\mathsf{PolyPS}}(p-1)$. Those parameters can be heuristically deduced from Table 2. For simplicity, we assume that $\lceil \log_p P \rceil = \log_p P$.

We first recall the differences of FDD in the binary and collapsed multiplexer tree functional bootstrapping. In the binary case, we perform $\log P$ SlotsMVB on the plaintext space $\mathbb{Z}_2$. As no other function than the mod_2 function needs to be evaluated, those are equivalent to AKP-FBT. Also, we can save one call to SlotsToCoeffs and ModSwitch for the last digit, as we do not need to convert it back to RLWE. This leads to the following expression of the approximate runtime of FDD in BMT-FBT

$$\mathcal{C}_{\mathsf{PreBMT}} = (\alpha + \beta) \log P - \gamma.$$

In the collapsed case, we perform $\lceil \log_p P \rceil$ SlotsMVB on $\mathbb{Z}_p$ with width $p+1$ to evaluate the p Kronecker deltas and the mod_p function, for a small p. Here, we can also save the last SlotsToCoeffs and ModSwitch. The approximate runtime can be expressed as

$$\mathcal{C}_{\mathsf{PreCMT}}(p) = \big(\alpha \log p + \beta + p \cdot \mathcal{C}_{\mathsf{PolyPS}}(p-1)\big) \log_p P - \gamma.$$

In the previous expression, $C_{\mathsf{PolyPS}}(p-1)$ corresponds to the cost of PolyPS for a polynomial of degree $p-1$. Then, to compare the binary and collapsed methods, we want to know whether the inequality $\mathcal{C}_{\mathsf{PreBMT}} > \mathcal{C}_{\mathsf{PreCMT}}(p)$ holds.

We can simplify the inequality of interest to:

$$\mathcal{C}_{\mathsf{PreBMT}} - \mathcal{C}_{\mathsf{PreCMT}}(p) > 0 \iff \big(\beta(\log p - 1) - p \cdot \mathcal{C}_{\mathsf{PolyPS}}(p-1)\big) \log_p P > 0.$$

For $P \geq 2$, this is equivalent to:

$$\big(\beta(\log p - 1) - p \cdot \mathcal{C}_{\mathsf{PolyPS}}(p-1)\big) > 0.$$

As one could expect, the inequality will not hold for large p, as the cost of the MVBs in FDD will become too substantial (concretely, $p \cdot \mathcal{C}_{\mathsf{PolyPS}}(p-1)$), as compared to the saved cost in p-independent bootstrapping operations $\beta(\log p - 1)$. Experimentally, we confirm that the previous inequality holds for $2 \leq \log p \leq 5$.

Regarding the cleaning that needs to be done after FDD, BMT-FBT always fares better than CMT-FBT. In the binary case, we need to perform cleaning on $\log P$ ciphertexts, whereas in the collapsed case we need to perform it on $p \cdot \log_p P = \frac{p}{\log p} \cdot \log P$ ciphertexts. Nevertheless, the cost of the cleaning is relatively small compared to the cost of FDD.

Multithreaded case. For parallel execution, we showed in Section 3.1 that MVB has the same cost as AKP-FBT, meaning that we can simplify the runtime expressions to

$$\mathcal{C}^{//}_{\mathsf{PreBMT}} = (\alpha + \beta)\log P - \gamma, \quad \mathcal{C}^{//}_{\mathsf{PreCMT}}(p) = (\alpha \log p + \beta)\log_p P - \gamma.$$

We can then deduce that:

$$\mathcal{C}^{//}_{\mathsf{PreBMT}} - \mathcal{C}^{//}_{\mathsf{PreCMT}}(p) > 0 \Longleftrightarrow \beta(\log p - 1)\log P > 0.$$

For a fixed $p > 2$, this inequality always holds for $P > 1$, meaning that FDD in CMT-FBT always outperforms the one in BMT-FBT. Regarding the cost of cleaning, they are exactly equal for both algorithms as they can be done in parallel on all the digits or Kronecker delta functions.

6.2 Comparison of BMT and CMT

In both BMT and CMT, the most expensive computations for large LUTs—the scalar multiplications, additions, and most of the ciphertext multiplications—are performed with the same number of RNS limbs left. This means that the cost of both algorithms should be similar, especially for very large P. However, for intermediate values of P, which are the ones discussed in this paper, there can be a non-negligible advantage in using one method versus the other, which is discussed below.

Let us assume that we are working with a power-of-two number of digits, that is, that $\log_p P = 2^a$ and $\log P = 2^b$, with a, b positive integers with $b > a$. As both algorithms perform the same vector matrix multiplications at the end, we focus on the difference in the KProduct algorithm, considering that the cost of the computation of $\mathsf{ctDigitVec}_i$ in BMT with $\log P$ additions is negligible. Additionally, we do not account for the fact that the KProduct algorithm needs more multiplicative depth in the binary case, leading to an increased number of RNS limbs and hence, more expensive homomorphic operations.

In the BMT algorithm, each call to KProduct requires a number of ciphertext multiplications equal to

$$\mathcal{N}_{\mathsf{BMT}} = \sum_{i=1}^{\log(\log\sqrt{P})} \log\sqrt{P} \cdot \frac{2^{2^i}}{2^i} = \sum_{i=1}^{b-1} 2^{b-(i+1)+2^i}.$$

Similarly for CMT, we have

$$\mathcal{N}_{\mathsf{CMT}} = \sum_{i=1}^{\log\lceil\log_p\sqrt{P}\rceil} \lceil\log_p\sqrt{P}\rceil \cdot \frac{p^{2^i}}{2^i} = \sum_{i=1}^{a-1} 2^{a-(i+1)} \cdot p^{2^i}.$$

Using the fact that $\log p = 2^{b-a}$, we get

$$\mathcal{N}_{\mathsf{CMT}} = \sum_{i=1}^{a-1} 2^{a-(i+1)} \cdot (2^{2^{b-a}})^{2^i} = \sum_{i=1}^{a-1} 2^{a-(i+1)+2^{b-a+i}}.$$

By changing the index of the sum $i = j - (b - a)$, we simply the expression to

$$\mathcal{N}_{\mathsf{CMT}} = \sum_{j=b-a+1}^{b-1} 2^{b-(j+1)+2^j}.$$

Subtracting both amounts, we get that each BMT's KProduct performs additional ciphertext multiplications

$$\begin{aligned}\mathcal{N}_{\mathsf{BMT}} - \mathcal{N}_{\mathsf{CMT}} = \sum_{i=1}^{b-1} 2^{b-(i+1)+2^i} - \sum_{i=b-a+1}^{b-1} 2^{b-(i+1)+2^i} &= \sum_{i=1}^{b-a} 2^{b-(i+1)+2^i} \\ &= \sum_{i=1}^{\log(\log p)} \log\sqrt{P}\frac{2^{2^i}}{2^i}.\end{aligned}$$

In the general case, theoretical analysis is more complicated due to using the ceiling function twice in determining the number of iterations of KProduct, $\lceil \log \lceil \log_p P \rceil \rceil$. We present in Figure 2 the exact number of ciphertext multiplications performed. Although every curve follows the same general tendency, we observed that there can be some significant differences for different digit sizes. We noticed that for almost all LUT sizes, there always exists a decomposition basis such that CMT performs fewer ciphertext multiplications than BMT.

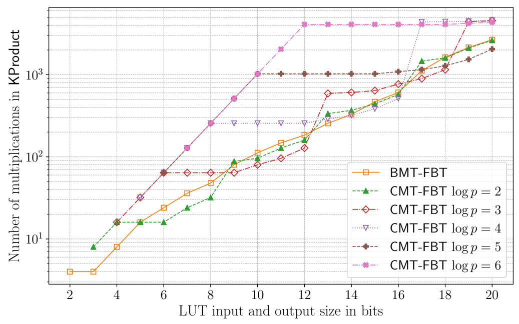


Figure 2: Total number of ciphertext multiplications performed in the two calls to KProduct for both the BMT and CMT algorithms, for different digit sizes p.

With perfect parallelization (which is less plausible than for FDD as the number of cores would depend on P), the ciphertext multiplications can be done in parallel at each level, meaning that CMT will always be more efficient than BMT because there are fewer levels as digits are larger.

7 Optimizations

We use AKP-FBT as a black-box and optimize the leveled computations, that is, the LUT computation based on multiplexer trees. We apply these optimizations to both the BMT-FBT and CMT-FBT algorithms.

Lazy relinearization and lazy rescaling. Rescaling and relinearization are costly operations, so we aim to minimize the number of these operations. To do so, we perform lazy rescaling and lazy relinearization. More precisely, we do not rescale ctResult$[i]$, but only rescale the result of the sum, returned by either the BMT or CMT algorithm. This saves roughly $\sqrt{P}$ rescaling operations.

Lazy relinearization requires more care, as without relinearization, the number of polynomials in a ciphertext increases exponentially with the number of multiplications. Moreover, as the cost of scalar multiplications is dominant for large LUTs, we want to have as few RNS limbs and polynomials as possible. Therefore, we always fully relinearize the ciphertexts that are going to be multiplied by the LUT (either ctKMSB or ctKMSD), and do not relinearize the last $O(\sqrt{P})$ multiplications of the other Kronecker product. Heuristically, this can be shown to be strictly better than the naive approach.

Optimal number of digits. The computational cost of Equation (5) is minimized when the number of digits, $n_d = \lceil \log_p P \rceil$, is a power of two. This provides two advantages. First, as an even number, it allows the digits to be split into two equal halves which balances the computation by ensuring the two Kronecker products yield vectors of identical size $p^{n_d/2} = p^{\lceil \log_p \sqrt{P} \rceil} = p^{\log_p P - \lceil \log_p \sqrt{P} \rceil}$. Second, a power-of-two structure enables the evaluation of each Kronecker product using a binary tree, minimizing both the total number of multiplications and the multiplicative depth. For this reason, a 4-digit decomposition provides a positive trade-off for CMT-FBT, enabling an efficient multiplexer tree while keeping the functional digit decomposition costs low.

Parallelization. In the functional digit decomposition, we parallelize the different polynomial evaluations of PolyPS in the MVB, achieving a latency similar to the original digit decomposition algorithm. We also parallelize the KProduct algorithm, as well as the last **for** loop in BMT and CMT to process each block independently.

8 Experimental performance evaluation

We first describe the experimental setup. All benchmarks were run on a server with two AMD EPYC 7302 16-Core Processor clocked at 3.0GHz and 128GB of RAM. The server has Ubuntu 22.04.5 LTS and uses OpenFHE v1.2.0 compiled with clang++ 14 (with NATIVEOPT on). We use the FIXEDMANUAL scaling and hybrid key switching (HKS).

We use the ring dimension $N = 2^{16}$ for large LUT evaluation and increase it to $N = 2^{17}$ for very large LUT (above 18 bits). We pack N values in the RLWE ciphertext and make use of both the real and imaginary parts of the CKKS ciphertext. We use sparse secret keys with a Hamming weight h of 192. Given the ring dimension N, we use the maximum CKKS modulus $Q'_L P'$ threshold that achieves 128 bits of security. As in [AKP24], for $N = 2^{16}$ we set the $Q'_L P'$ threshold to 1553 bits and for $N = 2^{17}$, to 3104 bits.

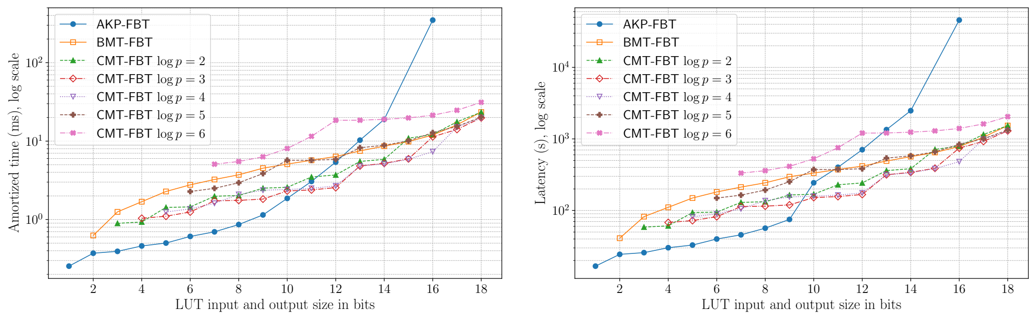
For an accurate comparison with direct functional bootstrapping, we ran the same benchmarks with the same implementation and parameters as in [AKP24], on our machine. Runtimes are provided in Appendix B. Note that for LUTs of sizes over 14 bits, we had to use 128-bit size words instead of 64-bit. The LUT values we used in our experiments are randomly sampled in their output space.

The parameters are chosen to provide the best amortized time while still respecting the modulus threshold for 128 bits of security. The reported times are for the optimized algorithms and the best respective decomposition basis.

8.1 BMT-FBT and CMT-FBT implementation and performance

The performance of CMT-FBT depends on the choice of the decomposition basis for a given target LUT size, as previously seen in Sections 5 and 7. To back our theoretical findings, we test CMT-FBT for different LUT sizes and digit sizes. In Figure 3, we depict the single-threaded amortized time and latency for evaluating an LUT under different digit sizes to find the most efficient one. We can see that CMT-FBT outperforms AKP-FBT in terms of amortized time starting with 11-bit LUTs, and consistently outperforms it until

16 bits, which is the current limit for AKP-FBT (BMT-FBT outperforms it starting with 13-bit LUTs). In terms of latency, because we work on a twice smaller ring (for up to 18-bit LUTs), we get an additional speed-up factor of two, and both CMT-FBT and BMT-FBT start outperforming AKP-FBT from 10 bits. Regarding the optimal decomposition basis, we see that $\log p = 3$ is the optimal digit size for $10 \leq \log P \leq 12$, that is, for four 3-bit digits, but is then outperformed by $\log p = 4$ for $13 \leq \log P \leq 16$, that is, for four 4-bit digits. For LUTs larger than 16 bits, the amortized time and latency of BMT-FBT and CMT-FBT (for all digit sizes p) get closer to each other. This is explained by the fact that the P scalar ciphertext multiplications become the dominant part of the computation, which is the same for both algorithms and all practical digit sizes.



(a) Amortized time (b) Latency

Figure 3: Comparison of the single-threaded amortized time (a) and latency (b) of a random LUT evaluation on a random input using AKP-FBT, BMT-FBT and CMT-FBT for different digit sizes (less is better).

In Figures 3a and 3b, we also see an increase in amortized time from 12 to 13 bits on all of the curves for the BMT-FBT and CMT-FBT methods. This is because we start to require a higher precision for the binary digits/Kronecker delta functions to achieve correctness, that is, we have to evaluate the cleaning polynomial $3X^2 - 2X^3$ on the ciphertexts resulting from FDD. This additional evaluation consumes two multiplicative levels. Hence, to remain below the threshold for the largest modulus, we have to reduce the levels available for homomorphic encoding and decoding (SlotsToCoeffs and CoeffsToSlots), and the number of HKS auxiliary limbs for large LUTs, resulting in performance degradation.

Table 3 shows the latency and amortized time of CMT-FBT for the best decomposition basis. Although our method is 2.0 times slower for 8-bit LUTs compared to AKP-FBT, we are 1.28 times faster for 11-bit LUTs and up to 47.5 times faster for 16-bit LUTs.

8.2 Multi-threaded performance

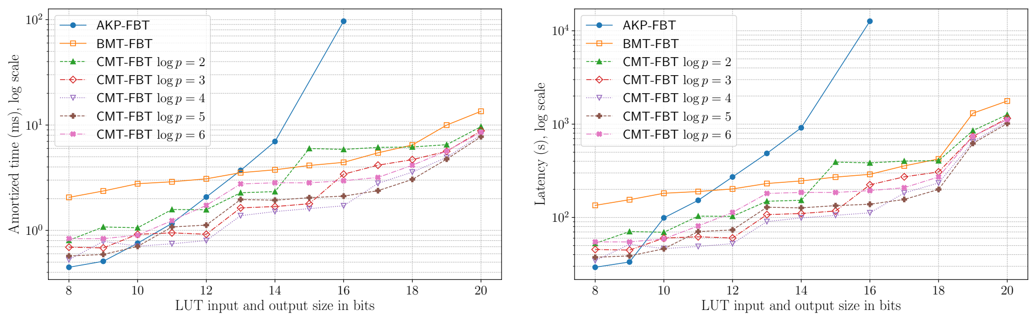
We now describe our multi-threaded experiments. As CMT-FBT is parallelizable, we expect an even better speed-up than in the single-threaded case. We report experimental results using parallelization on the same hardware, on which we ran the same benchmarks as in Section 8.1. In Figure 4, we observe a similar trend as in the single-threaded setting: AKP-FBT looks non-linear in the size of the LUT in bits, whereas CMT-FBT is almost linear. We indeed see that parallelization benefits our method, as we start outperforming AKP-FBT in both latency and amortized time from 10 bits and onward.

From Figure 4, we observe that larger digit sizes, especially $\log p = 6$, are more advantageous in a parallelized setting, as the MVBs in FDD parallelize well. We can also see "step" increases in the curves of CMT-FBT, when the number of digits $\lceil \log_p P \rceil$ increases.

We provide the runtime of CMT-FBT for the best decomposition basis, as well as the speed-up compared to (parallelized) AKP-FBT in Table 4.

Table 3: Single-threaded runtime measurements for a random LUT on $\mathbb{Z}_P$, on an encrypted random input using CMT-FBT. Q is the initial ciphertext modulus before digit decomposition; q, the modulus used in CKKS leveled computations; and $\log(Q'_L P')$, the number of bits in the largest CKKS modulus. A single limb is left after CMT-FBT. The speed-up factor of CMT-FBT compared to AKP-FBT is computed as $\mathcal{S} = T_{\textsf{AKP-FBT}}/T_{\textsf{CMT-FBT}}$.

$\log P$	$\log p$	$\log Q$	$\log q$	N	$\log(Q'_L P')$	Latency (s)	Amtz. time (ms)	$\mathcal{S}$
8	3	53	48	2^{16}	1452	115.02	1.76	0.49
10	3	55	48	2^{16}	1404	151.71	2.31	0.80
11	3	56	48	2^{16}	1404	156.38	2.39	1.28
12	3	57	48	2^{16}	1404	167.27	2.55	2.11
13	4	57	48	2^{16}	1452	315.04	4.81	2.14
14	4	58	48	2^{16}	1452	335.88	5.13	3.69
16	4	60	48	2^{16}	1452	480.80	7.34	47.5
18	4	62	48	2^{16}	1500	1,290.5	19.69	—
20	4	75	59	2^{17}	2074	8,447.0	64.45	—



(a) Amortized time (b) Latency

Figure 4: Comparison of the multi-threaded amortized time (a) and latency (b) of a random LUT evaluation on a random input using AKP-FBT, BMT-FBT and CMT-FBT for different digit sizes (less is better).

For very large LUTs (19 bits and above), we need to increase the ring dimension to have the necessary multiplicative depth to perform multiple rounds of cleaning[5] on the output of FDD, as well as increase the scaling factor. We were able to evaluate 20-bit-to-20-bit LUTs, which, to our knowledge, is far beyond what has currently been achieved for random, unstructured LUT. Our method allows the evaluation of even larger LUTs, but 20-bit LUTs is the limit for our server with 128GB of RAM. To target larger LUTs (with 22 or 24 bits), one should either increase the RAM or change the CMT algorithm to store only one ciphertext per level of the multiplexer tree, at the expense of additional multiplicative depth and slower computations.

8.3 Memory usage

For LUTs of 10 bits in size and above, our method outperforms AKP-FBT also from the perspective of RAM consumption. This is mainly due to the fact that we have a ring dimension that is twice as small, which leads to smaller ciphertexts and smaller evaluation keys (and we can fit all computations in 64-bit native words). In particular, for 12-bit LUTs, we notice that RAM consumption is reduced from 27.4 GB to 13.9 GB. We present

[5]This refers to the sequential evaluation of $G_{\textsf{id}}$. At least two successive evaluations of the cleaning polynomial $G_{\textsf{id}}$ are necessary for LUTs strictly larger than 20 bits.

Table 4: Multi-threaded runtime measurements for a random LUT on $\mathbb{Z}_P$, on an encrypted random input using CMT-FBT. Notations are the same as in Table 3. The speed-up factor of CMT-FBT compared to AKP-FBT is computed as $\mathcal{S} = T_{\mathsf{AKP\text{-}FBT}}/T_{\mathsf{CMT\text{-}FBT}}$.

$\log P$	$\log p$	$\log Q$	$\log q$	N	$\log(Q'_L P')$	Latency (s)	Amtz. time (ms)	$\mathcal{S}$
8	4	52	48	2^{16}	1524	34.48	0.53	0.85
10	5	53	48	2^{16}	1452	46.17	0.70	1.07
12	4	56	48	2^{16}	1452	52.46	0.80	2.59
14	4	58	48	2^{16}	1404	98.94	1.51	4.62
16	4	60	48	2^{16}	1452	112.17	1.71	56.5
18	5	61	48	2^{16}	1500	199.69	3.05	—
20	5	74	59	2^{17}	2074	1,014.8	7.74	—

a summary of the maximum RAM consumption in Table 5.

Table 5: Comparison of peak RAM consumption for the evaluation of LUTs on $\mathbb{Z}_P$ in the single-threaded setting. For CMT-FBT, we use the same optimal digit size p as in Table 3. We highlight the smallest memory consumption in bold.

$\log P$	10	12	14	16
AKP-FBT Peak RAM (GB)	22.9	27.4	32.3	110.2
BMT-FBT Peak RAM (GB)	22.4	22.6	**24.8**	**26.2**
CMT-FBT Peak RAM (GB)	**13.8**	**13.9**	27.3	28.5

9 Concluding remarks

Building on the functional bootstrapping method AKP-FBT [AKP24], we propose two novel functional bootstrapping procedures: BMT-FBT and CMT-FBT. Our performance evaluation shows that CMT-FBT is always the most efficient option, which can practically evaluate arbitrary functions represented as LUTs of up to 20 bits.

Since we used CKKS functional bootstrapping in our procedures as a black box, CMT-FBT will benefit from further improvements in CKKS and CKKS (functional) bootstrapping, both in terms of overall complexity and parallelism. Exploring the reduction of memory consumption (affected both by the computation depth and the ring dimension) is also critical for achieving larger LUT sizes.

More generally, our work was developed by exploring both the common ground and differences between the FHE schemes DM/CGGI and CKKS, to optimally tailor the use of efficient homomorphic computations to each scheme. Recently, concepts once viewed as DM/CGGI-specific (functional bootstrapping [BCKS24, BKSS24, AKP24], blind rotation [CHKS25], and in our current work, variants of ciphertext multiplexer trees) have now permeated to CKKS, where their adapted forms deliver superior amortized efficiency. We believe that continuing this direction of cross-migrating concepts towards the evaluation of non-smooth fixed-point functions over large inputs is a promising avenue for research.

Finally, this work can serve as a practical starting point for RAM-FHE. The Kedlaya-Umans preprocessing and ideas introduced in [LMW23] should be further studied to develop practical solutions.

References

[AAB+22] Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish

Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022.

[AKP24] Andreea Alexandru, Andrey Kim, and Yuriy Polyakov. General functional bootstrapping using CKKS. Cryptology ePrint Archive, Paper 2024/1623, 2024. To appear in CRYPTO 2025.

[BBB$^+$23] Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Parameter optimization and larger precision for (T)FHE. *J. Cryptol.*, 36(3):28, 2023.

[BCC$^+$25] Jean-Philippe Bossuat, Rosario Cammarota, Ilaria Chillotti, Benjamin R. Curtis, Wei Dai, Huijing Gong, Erin Hales, Duhyeong Kim, Bryan Kumara, Changmin Lee, Xianhui Lu, Carsten Maple, Alberto Pedrouzo-Ulloa, Rachel Player, Yuriy Polyakov, Luis Antonio Ruiz Lopez, Yongsoo Song, and Donggeon Yhee. Security guidelines for implementing homomorphic encryption. *IACR Communications in Cryptology*, 1(4), 2025.

[BCKS24] Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, and Damien Stehlé. Bootstrapping bits with ckks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 94–123. Springer, 2024.

[BCMR24] Foteini Baldimtsi, Kostas Kryptos Chalkias, Varun Madathil, and Arnab Roy. Sok: Privacy-preserving transactions in blockchains. *Cryptology ePrint Archive*, 2024.

[BGGJ19] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. Simulating homomorphic evaluation of deep learning predictions. In Shlomi Dolev, Danny Hendler, Sachin Lodha, and Moti Yung, editors, *Cyber Security Cryptography and Machine Learning*, pages 212–230, Cham, 2019. Springer International Publishing.

[BGP$^+$20] Marcelo Blatt, Alexander Gusev, Yuriy Polyakov, Kurt Rohloff, and Vinod Vaikuntanathan. Optimized homomorphic encryption solution for secure genome-wide association studies. *BMC Medical Genomics*, 13(83), 2020. https://eprint.iacr.org/2019/223.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[BK25] Dan Boneh and Jaehyung Kim. Homomorphic encryption for large integers from nested residue number systems. Cryptology ePrint Archive, Paper 2025/346, 2025. To appear in CRYPTO 2025.

[BKSS24] Youngjin Bae, Jaehyung Kim, Damien Stehlé, and Elias Suvanto. Bootstrapping small integers with ckks. In *Advances in Cryptology – ASIACRYPT 2024*, pages 330–360. Springer, 2024.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Annual cryptology conference*, pages 868–886. Springer, 2012.

[BSS$^+$22] Adda Akram Bendoukha, Oana Stan, Renaud Sirdey, Nicolas Quero, and Luciano Freitas. Practical homomorphic evaluation of block-cipher-based hash functions with applications. In *International Symposium on Foundations and Practice of Security*, pages 88–103. Springer, 2022.

[BTPH22] Jean-Philippe Bossuat, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In *International Conference on Applied Cryptography and Network Security*, pages 521–541. Springer, 2022.

[CBSZ23] Pierre-Emmanuel Clet, Aymen Boudguiga, Renaud Sirdey, and Martin Zuber. Combo: A novel functional bootstrapping method for efficient evaluation of nonlinear functions in the encrypted domain. In *Progress in Cryptology - AFRICACRYPT 2023*, 2023.

[CCP24] Jung Hee Cheon, Hyeongmin Choe, and Jai Hyun Park. Tree-based lookup table on batched encrypted queries using homomorphic encryption. Cryptology ePrint Archive, Paper 2024/087, 2024.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*, pages 3–33. Springer, 2016.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[CHK$^+$18a] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part I 37*, pages 360–384. Springer, 2018.

[CHK$^+$18b] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*, pages 347–368. Springer, 2018.

[CHKS25] Jung Hee Cheon, Guillaume Hanrot, Jongmin Kim, and Damien Stehlé. Ship: A shallow and highly parallelizable ckks bootstrapping algorithm. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 398–428. Springer, 2025.

[CIM19] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. New techniques for multi-value input homomorphic evaluation and applications. In *Cryptographers' Track at the RSA Conference*, pages 106–126. Springer, 2019.

[CJP21] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *Cyber Security Cryptography and Machine Learning: 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8–9, 2021, Proceedings 5*, pages 1–19. Springer, 2021.

[CKKL24] Heewon Chung, Hyojun Kim, Young-Sik Kim, and Yongwoo Lee. Amortized large look-up table evaluation with multivariate polynomials for homomorphic encryption. *Cryptology ePrint Archive*, 2024.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in cryptology–ASIACRYPT 2017: 23rd international conference on the theory and applications of cryptology and information security, Hong kong, China, December 3-7, 2017, proceedings, part i 23*, pages 409–437. Springer, 2017.

[DM15] Léo Ducas and Daniele Micciancio. Fhew: bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 617–640. Springer, 2015.

[DMKMS24] Gabrielle De Micheli, Duhyeong Kim, Daniele Micciancio, and Adam Suhl. Faster amortized fhew bootstrapping using ring automorphisms. In *IACR International Conference on Public-Key Cryptography*, pages 322–353. Springer, 2024.

[DMPS24] Nir Drucker, Guy Moshkowich, Tomer Pelleg, and Hayim Shaul. Bleach: cleaning errors in discrete computations over ckks. *Journal of Cryptology*, 37(1):3, 2024.

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

[GBA21] Antonio Guimarães, Edson Borin, and Diego F Aranha. Revisiting the functional bootstrap in tfhe. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 229–253, 2021.

[Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

[GPVL23] Antonio Guimarães, Hilder VL Pereira, and Barry Van Leeuwen. Amortized bootstrapping revisited: Simpler, asymptotically-faster, implemented. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 3–35. Springer, 2023.

[HHWW19] Ariel Hamlin, Justin Holmgren, Mor Weiss, and Daniel Wichs. On the plausibility of fully homomorphic encryption for rams. In *Annual International Cryptology Conference*, pages 589–619. Springer, 2019.

[Kim25] Jaehyung Kim. Efficient homomorphic integer computer from CKKS. Cryptology ePrint Archive, Paper 2025/066, 2025.

[KS22] Kamil Kluczniak and Leonard Schild. FDFB: Full domain functional bootstrapping towards practical fully homomorphic encryption. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022.

[LMP22] Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. Large-precision homomorphic sign evaluation using fhew/tfhe bootstrapping. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 130–160. Springer, 2022.

[LMW23] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic ram computation from ring lwe. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, pages 595–608, 2023.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*, pages 1–23. Springer, 2010.

[LW23] Zeyu Liu and Yunhao Wang. Amortized functional bootstrapping in less than 7 ms, with o˜(1) polynomial multiplications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 101–132. Springer, 2023.

[MS18] Daniele Miccianco and Jessica Sorrell. Ring Packing and Amortized FHEW Bootstrapping. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 100:1–100:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[PS73] Michael S Paterson and Larry J Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing*, 2(1):60–66, 1973.

[TBC$^+$25] Daphné Trama, Aymen Boudguiga, Pierre-Emmanuel Clet, Renaud Sirdey, and Nicolas Ye. Designing a general-purpose 8-bit (t) fhe processor abstraction. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(2):535–578, 2025.

[WHS$^+$24] Ruida Wang, Jincheol Ha, Xuan Shen, Xianhui Lu, Chunling Chen, Kunpeng Wang, and Jooyoung Lee. Refined TFHE leveled homomorphic evaluation and its application. Cryptology ePrint Archive, Paper 2024/1318, 2024. To appear in ACM CCS 2025.

[WL24] Benqiang Wei and Xianhui Lu. Improved homomorphic evaluation for hash function based on tfhe. *Cybersecurity*, 7(1):14, 2024.

A RNS CKKS operations

We introduce the operations on RNS CKKS ciphertexts which we used in our implementation. We assume that the ciphertexts or plaintexts have the same modulus. If the modulus Q'_ℓ differ, we first perform $\mathsf{DropLevels}$ to drop the ciphertext with the highest modulus to the same level as the ciphertext with the smallest modulus.

- $\mathsf{KeySwitchGen}(s, s_{\mathsf{new}})$: For the gadget modulus P' and a decomposition base w, both power of two, let $d_{\mathsf{swk}} = \lceil \log_w(Q'_L) \rceil$. Sample uniformly at random $a'_j \leftarrow \mathcal{R}_{Q'_L P'}$ and errors $e'_j \leftarrow \chi_{\mathsf{err}}$ and return

$$\mathsf{swk} = \{(\mathsf{swk}_{j,0}, \mathsf{swk}_{j,1})\}_{j=0}^{d_{\mathsf{swk}}-1} \text{ with } \begin{cases} \mathsf{swk}_{j,0} = -a'_j \cdot s_{\mathsf{new}} + e'_j + P' w^j \cdot s \\ \mathsf{swk}_{j,1} = a'_j \end{cases}$$

  Set $\mathsf{rlk} = \mathsf{KeySwitchGen}(s, s^2)$ and $\mathsf{cnk} = \mathsf{KeySwitchGen}(s, s^{-1})$.

- $\mathsf{KeySwitch}(\mathsf{ct}, \mathsf{swk})$:For $\mathsf{ct} = (c_0, c_1) \in \mathcal{R}^2_{Q'_\ell}$ and $\mathsf{swk} = \{(\mathsf{swk}_{j,0}, \mathsf{swk}_{j,1})\}_{j=0}^{d_{\mathsf{swk}}-1}$, decompose c_1 into d_{swk} polynomial digits $\{c_{1,j}\}_{j=0}^{d_{\mathsf{swk}}}$ so that $c_1 = \sum_{j=0}^{d_{\mathsf{swk}}-1} w^j \cdot c_{1,j}$. Return

$$\left[\left(c_0 + \left\lfloor \frac{\sum_{j=0}^{d_{\mathsf{swk}}-1} c_{1,j} \cdot \mathsf{swk}_{j,0}}{P'} \right\rceil, \left\lfloor \frac{\sum_{j=0}^{d_{\mathsf{swk}}-1} c_{1,j} \cdot \mathsf{swk}_{j,1}}{P'} \right\rceil \right) \right]_{Q'_\ell}.$$

- $\mathsf{DropLevels}(\{\mathsf{ct}, Q'_\ell\}, k \leq \ell)$: return $\{\mathsf{ct}, Q'_{\ell-k}\}$.

- $\mathsf{AddConst}(\{\mathsf{ct}, Q'_\ell\}, z)$: For $\mathsf{ct} \in \mathcal{R}^2_{Q'_\ell}$ with scaling factor Δ and $z = a + ib \in \mathbb{C}$, set $\mathsf{ct}_{\mathsf{Add}} \leftarrow [\mathsf{ct} + (\lfloor \Delta \cdot (a + b \cdot X^{N/2}) \rceil, 0)]_{Q'_\ell}$ and return $\{\mathsf{ct}_{\mathsf{Add}}, Q'_\ell\}$

- $\mathsf{AddPlain}(\{\mathsf{ct}, Q'_\ell\}, \mathsf{pt})$: For $\mathsf{ct} \in \mathcal{R}^2_{Q'_\ell}$ and $\mathsf{pt} \in \mathcal{R}$, set $\mathsf{ct}_{\mathsf{Add}} \leftarrow [\mathsf{ct} + (\mathsf{pt}, 0)]_{Q'_\ell}$ and return $\{\mathsf{ct}_{\mathsf{Add}}, Q'_\ell\}$

- $\mathsf{Add}(\{\mathsf{ct}_1, Q'_\ell\}, \{\mathsf{ct}_2, Q'_\ell\})$: For $\mathsf{ct}_1, \mathsf{ct}_2 \in \mathcal{R}^2_{Q'_\ell}$, set $\mathsf{ct}_{\mathsf{Add}} \leftarrow [\mathsf{ct}_1 + \mathsf{ct}_2]_{Q'_\ell}$ and return $\{\mathsf{ct}_{\mathsf{Add}}, Q'_\ell\}$.

- $\mathsf{MulConst}(\{\mathsf{ct}, Q'_\ell\}, z, \Delta')$: For $\mathsf{ct} \in \mathcal{R}^2_{Q'_\ell}$ with scaling factor Δ and $z = a + ib \in \mathbb{C}$, set $\mathsf{ct}_{\mathsf{Mul}} \leftarrow \lfloor \Delta' a \rceil \cdot \mathsf{ct} + \lfloor \Delta' b \rceil \cdot \mathsf{ct} \cdot X^{N/2}$ and return $\{\mathsf{ct}_{\mathsf{Mul}}, Q'_\ell\}$ with a new scaling factor $\Delta\Delta'$

- $\mathsf{MulInt}(\{\mathsf{ct}, Q'_\ell\}, \lambda)$: For $\mathsf{ct} \in \mathcal{R}^2_{Q'_\ell}$ with scaling factor Δ and $\lambda \in \mathbb{Z}$, set $\mathsf{ct}_{\mathsf{Mul}} \leftarrow k \cdot \mathsf{ct}$ and return $\{\mathsf{ct}_{\mathsf{Mul}}, Q'_\ell\}$.

- $\mathsf{MulPlain}(\{\mathsf{ct}, Q'_\ell\}, \mathsf{pt})$: For $\mathsf{ct} = (c_0, c_1) \in \mathcal{R}^2_{Q'_\ell}$ with scaling factor Δ and $\mathsf{pt} \in \mathcal{R}$ with scaling factor Δ', set $\mathsf{ct}_{\mathsf{Mul}} \leftarrow (c_0 \cdot \mathsf{pt}, c_1 \cdot \mathsf{pt})$ and return $\{\mathsf{ct}_{\mathsf{Mul}}, Q'_\ell\}$ with a new scaling factor $\Delta\Delta'$

- $\mathsf{Mul}(\{\mathsf{ct}_1, Q'_\ell\}, \{\mathsf{ct}_2, Q'_\ell\})$: For $\mathsf{ct}_1, \mathsf{ct}_2 \in \mathcal{R}^2_{Q'_\ell}$ with $\mathsf{ct}_i = (b_i, a_i)$ and scaling factor Δ_i, compute $(d_0, d_1, d_2) = [(b_1 b_2, a_1 b_2 + a_2 b_1, a_1 a_2)]_{Q'_\ell}$, set $\mathsf{ct}_{\mathsf{Mul}} \leftarrow [(d_0, d_1) + \mathsf{KeySwitch}(d_2, \mathsf{rlk})]_{Q'_\ell}$ and return $\{\mathsf{ct}_{\mathsf{Mul}}, Q'_\ell\}$ with a new scaling factor $\Delta_1\Delta_2$.

- $\mathsf{Rescale}(\{\mathsf{ct}, Q'_\ell\})$:For $\mathsf{ct} \in \mathcal{R}^2_{Q'_\ell}$ with scaling factor Δ, set $\mathsf{ct}_{\mathsf{Rscl}} \leftarrow \lfloor {q'}_\ell^{-1} \cdot \mathsf{ct} \rceil$ with ${q'}_\ell^{-1} \in \mathbb{R}$ and return $\{\mathsf{ct}_{\mathsf{Rscl}}, Q'_{\ell-1}\}$ with a new scaling factor Δ / q_ℓ.

- $\mathsf{Conj}(\{\mathsf{ct}, Q'_\ell\})$:For $\mathsf{ct} = (c_0, c_1) \in \mathcal{R}^2_{Q'_\ell}$, set:

$$\mathsf{ct}_{\mathsf{Conj}} \leftarrow [(c_0^{-1}, 0) + \mathsf{KeySwitch}(c_1, \mathsf{cnk})]_{Q'_\ell}$$

For completeness, we also detail the CKKS bootstrapping procedure.

- ModRaise: We raise the modulus from q'_0 to the base modulus Q'_L, at which we have multiplicative depth L. That is, for an exhausted encryption of a plaintext $m(X)$ with modulus q'_0, we obtain an encryption of $t(X) = m(X) + q'_0 \cdot I(X)$ with $I \in \mathbb{Z}[X]$.
- CoeffsToSlots: Although we regained modulus, the plaintext is not what we started with. To get rid of $q'_0 \cdot I(X)$, we start by homomorphically applying the encoding procedure τ'^{-1} to the coefficients of $t(X)$ to move them to the slots representation.
- EvalMod: Then we evaluate the modulo function $x \mapsto x \mod q'_0$ on each slot to remove $q'_0 \cdot I(X)$. To do so, we usually use a polynomial interpolation of $x \mapsto q'_0/(2\pi)\sin(2\pi x/q'_0)$, which is a good approximation of $\mod q'_0$ when $q'_0 \gg m_i$. After homomorphically evaluating this polynomial, we obtain $\hat{m}_i \approx m_i$ in each slot.
- SlotsToCoeffs: We homomorphically run the decoding procedure, that is τ', to put the result back in the coefficients, thus obtaining an encryption of $\hat{m}(X)$.

The bootstrapping procedure consumes multiplicative levels for homomorphic encoding, decoding and polynomial evaluation. Therefore, the resulting ciphertext has as depth $L - L_{\mathsf{CoeffsToSlots}} - L_{\mathsf{EvalMod}} - L_{\mathsf{SlotsToCoeffs}}$. We usually look for a good tradeoff in the precision of $\hat{m}(X)$ and the number of levels consumed during bootstrapping.

B Performance of AKP-FBT

For the AKP-FBT runtimes in Table A1 we use almost the same parameter set as in [AKP24] for an objective comparison. We had to increase Q for $\log p = 14$ by one bit to guarantee correct decryption. This is because we evaluate random LUTs, which have more discontinuities, compared to the mod_p function that was targeted in [AKP24].

Table A1: Single-threaded runtime measurements for a random LUT on $\mathbb{Z}_p$, on an encrypted random input using AKP-FBT. Q is the initial ciphertext modulus; and $\log(Q'_L P')$, the number of bits in the largest CKKS modulus. We include our best amortized times of BMT-FBT (for $\log p = 2$ and $\log p = 3$) and CMT-FBT (given in Table 3) for comparison.

$\log p$	$\log Q$	N	$\log(Q'_L P')$	Latency (s)	Amtz. time (ms)	Our amtz. time (ms)
1	35	2^{16}	870	16.685	0.2546	—
2	35	2^{16}	1035	24.392	0.3722	0.6260
3	37	2^{16}	1114	25.768	0.3932	1.2518
4	38	2^{16}	1234	30.181	0.4605	0.9305
5	43	2^{16}	1392	32.871	0.5016	1.1025
6	44	2^{16}	1460	39.933	0.6093	1.2479
7	46	2^{16}	1404	45.681	0.6970	1.6095
8	47	2^{16}	1535	56.593	0.8635	1.7550
9	48	2^{16}	1548	75.036	1.1450	1.8241
10	53	2^{17}	2077	243.95	1.8612	2.3149
11	53	2^{17}	2243	401.51	3.0632	2.3862
12	55	2^{17}	2420	706.98	5.3938	2.5523
13	58	2^{17}	2574	1,347.9	10.284	4.8071
14	59	2^{17}	2726	2,479.2	18.915	5.1252
16	80	2^{17}	3594	45,722	348.83	7.3365

In this work, we used AKP-FBT as a black box, but note that some of the implementation optimizations we introduced for our method can also be applied to AKP-FBT.

To show the multiple limitations of AKP-FBT for larger LUTs, we include an evaluation for a 16-bit LUT. In this case, we had to compile OpenFHE with NATIVE_SIZE=128, as the modulus exceeds 64 bits. This is one of the reasons why the runtime is so large compared to the smaller LUTs that use NATIVE_SIZE=64. Note that $\log(Q'_L P')$ exceeds the modulus threshold for 128-bit of security, resulting in a reduced security. Full security would require increasing the ring dimension to 2^{18}, which in turn would increase the latency even further. Moreover, for the 16-bit LUT, we choose the mod_p function instead of a random LUT, as a highly discontinuous LUT would need even larger parameters.

C Extensions of CMT-FBT and BMT-FBT

We propose several extensions of CMT-FBT(that also apply to BMT-FBT):

$\mathbb{Z}_P$ to $\mathbb{C}$ LUT. We can support the evaluation of complex-valued LUTs by replacing the integer multiplication with a constant multiplication, which consumes an additional level.

Slot-independent LUT. Similarly, we can support a slot-independent LUT evaluation, that is, evaluate a different LUT on each slot of the RLWE ciphertext, by replacing integer multiplications with plaintext multiplications. In this case, we need to prepare P plaintexts, $\{\mathsf{pt}_i\}_{0\leq i<P}$. For each potential input value $i \in \mathbb{Z}_P$, the plaintext pt_i is constructed such that its j-th slot contains the value $f_j(i)$—the result of evaluating the specific LUT for the j-th slot at input i. Regarding overhead, the difference is that we replace P scalar multiplications with P plaintext multiplications, which are costlier in terms of both runtime and noise growth. This novel approach is especially useful for parallel computing.

MVB. Another extension of CMT-FBT is the support for multi-value bootstrapping. Just as with AKP-FBT, most of the procedure can be done once to evaluate $k > 1$ LUTs on the same input (or slot-independent LUTs). FDD and Kronecker products can be done once for all k LUTs. Then, we need to perform P scalar multiplications, $O(P)$ additions, $\sqrt{P}$ ciphertext multiplications, SlotsToCoeffs and ModSwitch for each k output ciphertexts.

Decomposed large digits. Another variation of CMT-FBT is to work with already decomposed digits. This becomes crucial when dealing with very large LUTs, as the magnitude of the outputs of the LUT is the primary source of noise growth, as seen in Section 4. For instance, direct evaluation of an arbitrary 32-bit LUT would not be possible with CMT-FBT. A workaround is to decompose the 32-bit input into two 16-bit digits. We then use (MVB) CMT-FBT to compute the 2^{16} Kronecker deltas for each digit. We can then use those Kronecker delta functions to select the value of the result's LSD and MSD. In a sense, we can use CMT-FBT recursively.

Encrypted LUT. Similarly to the slot-independent LUTs evaluation, CMT-FBT can be used to evaluate encrypted (slot-independent) LUTs. This can be used to obfuscate computations, but it has a significant cost in memory as we need to store P ciphertexts.

D Noise analysis of CMT-FBT

In this section, we formalize the noise analysis of the CMT-FBT procedure. A similar analysis can be derived for BMT-FBT.

The CMT-FBT procedure involves several distinct stages, each impacting ciphertext noise differently:

- Functional Digit Decomposition: An initial RLWE ciphertext is iteratively decomposed. At each step, AKP-FBT is used to produce a CKKS ciphertext.
- Noise Cleaning: An optional leveled CKKS computation is performed on CKKS ciphertexts encrypting binary values to further reduce the noise.
- Collapsed Multiplexer Tree: A leveled CKKS computation to compute the LUT, which is the main source of noise growth.
- CKKS–RLWE Conversion: A final CKKS to RLWE conversion. The resulting error of the RLWE ciphertexts depends on the initial CKKS noise, as well as the error introduced during the conversion process.

As a precise bit-by-bit analysis would be exceedingly complex, we make the following simplifying assumptions based on [AKP24]:

- (1) Noise Reduction of AKP-FBT: The core of FDD is AKP-FBT. As established in [AKP24], its use of Hermite interpolation provides noise reduction capabilities. Concretely, for an RLWE ciphertext with an error below a given threshold, AKP-FBT outputs a CKKS ciphertext with a small, bounded noise, which we denote as $\varepsilon_{\mathsf{AKP}}$. This acts as a periodic noise resetting procedure for the digit ciphertexts. Additionally, the separate evaluation of the noise cleaning polynomial G_{id} [DMPS24] can be used to further reduce the magnitude of this noise and reduce noise magnitude by several bits.
- (2) Negligible Inter-Step Noise: We assume the error introduced by RLWE operations (subtraction, modulus switching) within the FDD loop is small and does not accumulate in a way that compromises the input condition for the next functional bootstrapping step.

Theorem 1 (Correctness of CMT-FBT)**.** *Let ct_{in} be an RLWE ciphertext with an error satisfying the input requirements of AKP-FBT for noise cleaning. Let the CMT-FBT parameters (P, p) and the CKKS and RLWE parameters be chosen to ensure 128-bit security and a sufficient multiplicative depth for the computation. The CMT-FBT procedure correctly evaluates a P-to-P LUT for a scaling factor Δ satisfying*

$$\Delta = \Omega(P^3 \lceil \log_p P \rceil).$$

Proof. The FDD algorithm executes $n_d = \lceil \log_p P \rceil$ iterations. For each iteration, SlotsMVB is called. Based on hypotheses (1) and (2), each of the $p \cdot n_d$ Kronecker delta ciphertexts produced by FDD has a well-controlled noise in the CKKS domain bounded by $\varepsilon_{\mathsf{AKP}}$:

$$|\varepsilon_{\mathsf{FDD}}| \leq \varepsilon_{\mathsf{AKP}}.$$

Then, the CMT procedure is the primary source of noise growth. As previously analyzed in Section 5.1, those computations result in a worst-case noise growth given by

$$|\varepsilon_{\mathsf{CMT}}| = O\big(P^2 \lceil \log_p P \rceil\big).$$

For the final conversion from CKKS to RLWE to be correct, the integer part of the message must be preserved. Given that the RLWE plaintext modulus is P, this translates to

$$\frac{\Delta}{2P} = \Omega\big(P^2 \lceil \log_p P \rceil\big).$$

This leads to the final condition on Δ:

$$\Delta = \Omega(P^3 \lceil \log_p P \rceil\big).$$

□