BLEACH: Cleaning Errors in Discrete Computations over CKKS

Nir Drucker, Guy Moshkowich, Tomer Pelleg and Hayim Shaul

IBM Research - Israel, Haifa, Israel.

Abstract

Approximated homomorphic encryption (HE) schemes such as CKKS are commonly used to perform computations over encrypted real numbers. It is commonly assumed that these schemes are not "exact" and thus they cannot execute circuits with unbounded depth over discrete sets, such as binary or integer numbers, without error overflows. These circuits are usually executed using BGV and B/FV for integers and TFHE for binary numbers. This artificial separation can cause users to favor one scheme over another for a given computation, without even exploring other, perhaps better, options. We show that by treating step functions as "clean-up" utilities and by leveraging the SIMD capabilities of CKKS, we can extend the homomorphic encryption toolbox with efficient tools. These tools use CKKS to run unbounded circuits that operate over binary and small-integer elements and even combine these circuits with fixed-point real numbers circuits. We demonstrate the results using the Turing-complete Conway's Game of Life. In our evaluation, for boards of size 128×128, these tools achieved an order of magnitude faster latency than previous implementations using other HE schemes. We argue and demonstrate that for large enough real-world inputs, performing binary circuits over CKKS, while considering it as an "exact" scheme, results in comparable or even better performance than using other schemes tailored for similar inputs.

Keywords: fully homomorphic encryption, encrypted binary circuits, CKKS, mixed integer-floating point operations, game of life

1 Introduction

1.1 Non-Interactive Homomorphic Encryption

The proliferation of applications that perform computations over encrypted data is observed, for example, by the large number of proposed privacy-preserving machine learning (PPML) solutions, e.g., [2, 7, 8, 27, 29, 31, 38, 39, 44–47, 51]. These solutions rely on homomorphic encryption (HE) as well as a combination of HE with multi-party computations (MPC) protocols, such as garbled circuits (GCs), oblivious transfers (OTs) or secret sharing (SS).

When a solution involves HE, it can be categorized as a client-aided (e.g., GAZELLE [31], nGraph-HE [8]) or a non-client-aided solution (e.g., HeLayers [2], HEMET [39]), depending on whether or not it asks the client to assist in the computations. Client-aided solutions allow certain demands from the HE scheme to be relaxed. Such demands can be the need for costly bootstrapping operations or other ways to deal with the accumulated error that grows because of the encrypted computations or the use of non-polynomial functions approximation. For example, GAZELLE [31] and nGraph-HE [8] perform neural network inference operation under encryption, but ask the client to perform the non-polynomial ReLU activation functions. These designs often include MPC to hide the intermediate results from the client.

Using a client-aided solution removes many restrictions and improves latencies but pays the price in the cost of interactive sessions. This goes against the original purpose of using FHE, which is to remove all burdens from the users and use the cloud for the entire computation. Using client-aided solutions can also involve additional security risks as explained in [3] or as demonstrated by [36] who showed that it can simplify model-extraction attacks when considering constructions such as GAZELLE [31].

There are two main challenges with non-interactive applications: latency and accuracy. The accuracy issue is solved using schemes such as BGV [12], B/FV [11, 22], and TFHE [18]; these use a bootstrapping operation, which is considered a costly operation. However, in CKKS, the situation is different. First, the CKKS design [15] operates over floating-point elements and assumes that the scheme noise can be blended with the noise of the floating-point plaintext computations; this means just a slight noise overhead of up to 1-bit per multiplication compared to plaintext floating-point operations. Second, the bootstrapping operation in CKKS is considered fast because it serves a different purpose than in BGV and TFHE. Here, the goal of the bootstrapping is not to reduce noise, which is already mixed with the plaintext value, but to enable ciphertexts to be used in further computations. In practice, it may even increase the noise and therefore support circuits with limited depth.

Today, there is an artificial categorization of schemes to potential applications. If a developer wants to evaluate a binary circuit, they will probably go with TFHE or BGV. While for neural networks (NNs) that involves non-accurate floating-point coefficients, an approximate scheme such as CKKS may

be the default choice (e.g., as in [2, 39]). The depth of the potential computation also plays a major role in the scheme choice, where CKKS suffers due to its potential noise growth when considering deep-enough circuits. In that sense, CKKS cannot always be treated as a "truly" fully homomorphic encryption (FHE) scheme, because the accumulated noise makes the computation results unusable. Thus, our goal was to answer the following three research questions:

1. Can we efficiently evaluate *unbounded* binary circuits using CKKS?
2. Can we efficiently evaluate *unbounded* circuits that only operate over integer numbers (hereafter, integer circuits) using CKKS?
3. Can we efficiently evaluate Boolean and integer *unbounded* circuits combined with circuits over floating point or complex elements over CKKS?

We answer affirmatively to all of these questions. Specifically, for Question 1, in Section 4 we demonstrate a binary circuit and inputs for which CKKS is efficient. In Section 5 we demonstrate an unbounded Boolean-integer circuit for which CKKS is efficient and in Section 8 we provide several applications that combine integer and floating point elements over CKKS. We note that the term efficiency in the above questions is ambiguous. To avoid ambiguity, we consider a solution efficient if its latency, amortised latency, or throughput are comparable to or better than an equivalent solution with other FHE schemes that serve the same target. We briefly review the properties of HE schemes in Section 3.1.

Sign Functions. The HE Add and Mul operations depend on the underlying plaintext elements. For some schemes that operate over binary fields, these operations are equivalent to binary XOR and AND operations. However, in CKKS they represent computation over complex numbers. Consequently, in schemes such as TFHE, it is possible to simulate branches (`IF` statements) using e.g., MUX gates, which can be constructed from AND and XOR gates, and thus evaluating any binary circuit. In contrast, simulating branching in CKKS is hard. This usually involves high-degree polynomials to achieve decent accuracy.

One solution is to use a step function that returns 0 or 1 depending on the conditional input. The accuracy of this function depends on two input parameters, which are defined informally as the available input precision α and the required output precision β. Figure 1 uses the function $h_1(x) = -2x^3 + 3x^2$ for $x \in [-0.2, 1.2]$ [16] to demonstrate how a step function can be generated. It presents several curves, each made from a different number of compositions of $h_1(x)$. As the number of compositions increases, the multiplication depth increases and the accuracy parameters α, β decrease. There are different ways to generate these step functions and we elaborate more on them in Section 3.

In CKKS, there are different use cases for the step function, e.g., as a comparison or a sign function, see [16]. We propose a new way to look at this family of functions. Instead of considering them mathematical operations, we suggest using them as "clean-up" utilities. For example, assume that a ciphertext encrypts the plaintext value 1.0003245, which corresponds to the

binary value 1, the goal of the clean-up utility is to remove the most significant digits after the decimal point. We use the name "utility" to avoid confusion with the current bootstrapping operations of CKKS.

When considering binary circuits in CKKS, the inputs are in $x \in [0 - \epsilon, 0.5 - \alpha] \cup [0.5 + \alpha, 1 + \epsilon]$, for some small ϵ. Thus, to clean them, we can use a step function that maps values that are close to $0, 1$ closer to $0, 1$, respectively. This partially answers Question 1: can we evaluate unbounded binary circuits on CKKS? But, can we do it efficiently?

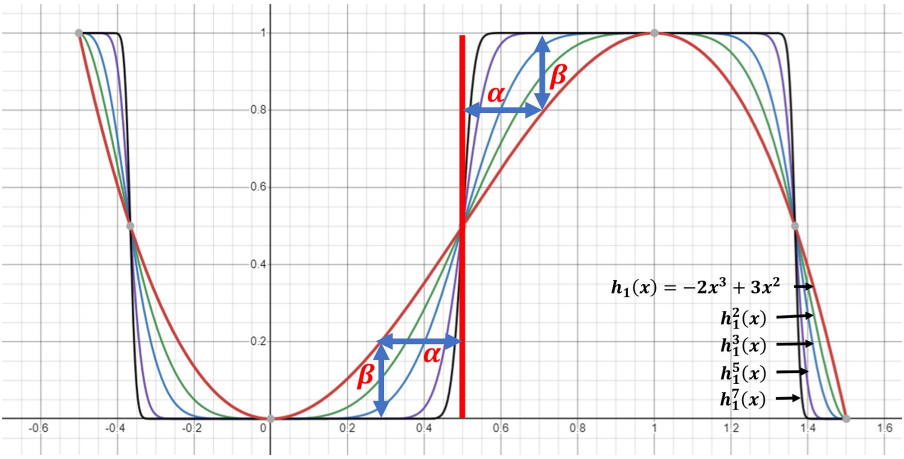


Figure 1: Different "step" function approximations achieved through using a different number of compositions of the function $h_1(x) = -2x^3 + 3x^2$ [16] over the input range $[-0.2, 1.2]$. The vertical red line is the middle (0.5) between the two step values 0 and 1, α is the distance of the input from that line to the curve for a given output accuracy β.

Parallelization. The answer to the above question depends on two factors: the choice of step function and the width of the circuit, i.e., its level of parallelization. The time it takes to clean up a binary bit depends on the performance of the step function. We argue that we can compensate for that time by using the single instruction multiple data (SIMD) property of CKKS, which allows us to perform $s = N/2$ operations in parallel for the polynomial ring parameter N. When we have a wide enough circuit that can leverage most of the slots of the input vector for CKKS, the speedup we achieve is so significant that the clean-up function performance only slightly affects the overall latency of the solution.

Interestingly, workloads with wide circuits that operate in parallel over a huge amount of data are not rare. In fact, they are the main target for optimization when considering cloud applications. The motivation for using the cloud lies in its computing power and its ability to process masses of information on behalf of the client. For these workloads, the latency of using

CKKS can be more efficient than other solutions that target binary circuits. In Section 5 we demonstrate our claim by using the recently published benchmark of the Game of Life that was originally demonstrated over TFHE [42, 43]. Our results show that for boards of 128×128 on a CPU with 8 cores, our CKKS solution achieves performance that is an order of magnitude faster than the original implementation.

We already identified one use case or domain of programs that today can run better on CKKS, i.e., the case of wide binary circuits. But we did not want to stop there. We extended our clean-up method to operate over integers by using a bit-extraction technique. When the input is known to be an integer in some range e.g., $[0, 2^8 - 1]$, and assuming that the error is still below some limit α, it is possible to perform bit-extraction over that integer, clean up every bit, and reconstruct the integer from its cleaned bits. The above clean-up method allows us to extend our answer to Question 3 to include integer or fixed-point workloads.

BLEACH. We called our clean-up methodology BLEACH. This methodology reduces the error accumulated inside ciphertexts of approximated HE schemes, when the underlying plaintexts are known to come from a discrete set of elements, e.g., binary or integer numbers. We name it BLEACH as it metaphorically "cleans" errors from data. Figure 2 schematically shows the BLEACH concept. We start from a standard HE scheme such as CKKS that gets as inputs Boolean, integer, and floating point elements. If during the computation we know that some ciphertexts contain elements from some discrete set of elements e.g., Booleans or integers, we reduce the error of the computation, which arrived from the scheme or polynomial approximations. Subsequently, we can use the cleaned ciphertexts in another round of encrypted computations. Before BLEACH, the error was accumulated in elements of all types and thus limited the depth of circuits that could possibly be executed. With BLEACH, the error is accumulated only in floating-point elements similar to mixed integer-floating point computations on standard CPUs.

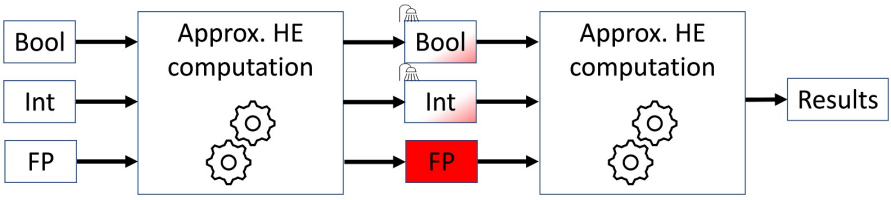


Figure 2: A schematic view of homomorphic encryption using the BLEACH methodology. Boolean and Integer elements are cleaned during the computation and thus allow deeper circuits or even unbounded circuits when only discrete set elements are involved. Red color indicates the accumulated error.

Our Contributions. Our contributions can be summarized as follows.

- We show that it is possible to efficiently perform unbounded binary circuits over CKKS and identify several cases where doing so is an order of magnitude faster than when using other schemes to perform the same circuit. This positions CKKS, and in general approximated FHE schemes, as a viable alternative for performing binary circuits.
- To showcase the performance advantage of using CKKS over arbitrarily deep circuits that combine binary circuits and integer operations, we implemented Conway's Game of Life. We show that for large boards e.g., 128×128 or larger, our CKKS implementation is more than an order of magnitude faster than the reference implementation used with TFHE.
- To enable deep integer circuits, we present a new error reduction algorithm for ciphertext encrypting integer numbers, which relies on decomposing the integer into its binary representation, reducing the error of its binary coefficients, and reconstructing it to get a ciphertext that encrypts the original integer with reduced error. We prove the correctness of our algorithm and discuss several applications where we can apply it. One example, is modular arithmetic over CKKS. Another interesting case combines a deep integer circuit with a floating-point circuit such as analytics-based decision trees or algorithms that require lookup tables. This combination can be done using only CKKS without involving other schemes.

Roadmap. The rest of the paper is organized as follows. Section 2 presents some additional related work. Section 3 lists the notation used in the paper, and the setup we used for the experiments. It also provides some background about optimal sign approximations and tile tensor based packing. We demonstrate the efficiency potential of executing binary circuits over CKKS in Section 4. Section 5 provides another demonstration for our cleanup paradigm that is based on the Game of Life. Section 6 presents our decompose method and some experimental results. Section 7 uses the decomposition method to perform integer arithmetic over CKKS and Section 8 presents some applications of this method. Finally, we discuss some interesting takeaways in Section 9 and conclude the paper in Section 10.

2 Related work

This section surveys several HE concepts related to our research questions, which we consider orthogonal topics. Specifically, we refer to works that evaluate the errors of a given circuit, perform bootstrapping operations, and move back and forth between HE schemes.

2.1 Approximation errors

Kim et.al [32] suggested variants for the original CKKS and RNS-CKKS schemes that reduce the approximation error. In their work, the amount of noise reduction depends on the circuit's operations and they can reduce the noise of encrypted floating point numbers. In contrast, BLEACH noise reduction is

independent of the circuit and enables noise to be reduced as low as needed. However, BLEACH can only reduce noise in ciphertexts that encrypts discrete numbers e.g., integers or binary numbers.

2.2 Bootstrapping in CKKS

Bootstrapping [25] for TFHE e.g., [17,41], BGV e.g., [14,28], and BF/V performs decryption of ciphertexts under HE and by that clean the accumulated error of a ciphertext during the circuit evaluation. In contrast, CKKS also supports Bootstrapping e.g., [6,9,13,30,35]. But here Bootstrapping does not remove the accumulated error, which comes from the basic CKKS operations, including the Bootstrapping itself, and from polynomial approximation errors. This is exactly what we wanted to remedy. Instead, it serves a different purpose, for efficiency reasons, the bootstrapping in CKKS only increases the modulus chain, which allows performing further computations on ciphertexts. However, Combining BLEACH with an efficient bootstrapping technique allows us to reduce error during the computation even after the scheme parameters are initialized.

Table 1 summarizes the latest bootstrapping capabilities for CKKS. We stress that BLEACH methodology is agnostic to the bootstrapping method and can be executed transparently over each one of the presented bootstrapping algorithms. Still, we present this table here, because BLEACH assumes a certain bound to the bootstrap error, and this data can be consumed from the table.

Table 1: CKKS SotA bootstrapping precision. Data is taken from [6].

Algorithm	Poly. degree	Number of slots	Bit precision
BMT+21 [9]	2^{15}	2^{14}	15.5
JM22 [30]	2^{16}	2^{3}	45
	2^{17}	2^{3}	100
LLK+22 [35]	2^{17}	2^{3}	100.11
	2^{17}	2^{12}	93.03
BCC+22 [6]	2^{15}	2^{14}	48
	2^{16}	2^{15}	255
	2^{17}	2^{16}	420

2.3 Bit decomposition for bootstrapping

Gentry et al. [26] introduced an improved bootstrapping implementation for BGV-like schemes by using a homomorphic computation method to extract the bits or digits of a message. This digit-extraction method was improved in [13] for BGV and for BF/V. Another bit extraction algorithm was introduced in [4] to deal with extracting integers from floating point elements in a BF/V like scheme. However, the methods of [4, 13, 26] are attached to BGV- and B/FV-like constructions that operate over fields or rings; they cannot be applied as-is to other schemes that operate over a complex plane such as CKKS.

To the best of our knowledge, our algorithm is the first concrete bit-extraction algorithm for CKKS. Furthermore, our algorithm serves a different purpose: we use it to remove the CKKS noise of each bit of an integer in the message. An advantage of our bit-extraction algorithm is that it relies on the basic CKKS operations, and does not require any modification for the scheme internals. Consequently, our bit-extraction algorithm can be used by other approximated HE schemes as-is.

2.4 Scheme switching

Scheme switching allows us to convert a ciphertext generated on one HE scheme to a ciphertext compatible with another scheme. For example, Pegasus [40] and Chimera [10] consider switching functions from CKKS to TFHE and vice versa. Chimera [10] also considers switching from BFV to TFHE and vice versa. Switching between BGV/BFV and CKKS is still considered an open question [5]. Scheme switching implementations are supposed to be implemented e.g., in OpenFHE [5] in the future.

Scheme switching is an interesting direction for performing integer cleanups in CKKS, as developers can take a "dirty" ciphertext, move it to TFHE, clean it using TFHE's bootstrapping, and return it to CKKS for an additional (possibly unlimited) number of operations. Nevertheless, because TFHE does not support SIMD operations, it is unclear whether the overall performance of cleaning a large e.g., 2^{15} number of elements in TFHE is worth the scheme switching complexity. Similarly, we could have used BGV or BF/V bootstrapping for the task. However, as mentioned in [5] these scheme switching methods are still under research.

Interestingly, previous studies showed that libraries or protocols that support many different schemes and complex state machine tend to cause more bugs or more vulnerabilities. For example, that is why the designers of TLS 1.3 [49] decided to narrow down the list of supported cryptographic primitives compared to TLS 1.2 [50]. It was also the reason that Google forked from OpenSSL and generated BoringSSL [1].

With the growth of the HE domain, we expect to see libraries of different types, generic libraries such as OpenFHE [5] that support many different schemes, and libraries that are dedicated to a specific scheme such as HEaaN [21] for CKKS and Zama's Concrete [19] for TFHE. We target the latter type of libraries that support only one scheme such as CKKS, and explore what workload characteristics a developer can expect to get from it.

3 Preliminaries and notation

We define Boolean elements with k bits of precision in CKKS as elements in the set $B_k = [0 - 2^{-k}, 0 + 2^{-k}] \cup [1 - 2^{-k}, 1 + 2^{-k}]$, i.e., Boolean elements with precision of k bits after the decimal point. This is the result of the CKKS error or the floating point approximation errors that are blended into the Boolean values. Similarly, we define integer numbers as values in $I_k = \bigcup_{n \in \mathbb{N}} [n - 2^{-k}, n + 2^{-k}]$.

Every integer number $n \in \mathbb{N}$, $n < 2^{N+1}$ has a binary representation $b = (b_N, \ldots, b_0)$ such that $n = \sum_{i=0}^{N} 2^i b_i$. We define $\mathrm{MSB}(x) = b_N$ and denote the i'th bit of x by $(x)_i = b_i$. In addition, an encryption of some value x is denoted with double brackets i.e., $[\![x]\!]$.

3.1 Homomorphic Encryption

Modern HE instantiations such as BGV [12], B/FV [11, 22], CKKS [15], and TFHE [18] rely on the hardness of the Ring-LWE problem or similar problems, and operate over rings of polynomials. They provide at least six methods: Gen, Enc, Dec, Add, Mul, and Rot. The secret, public, and evaluation keys are generated using the *Gen* method. The secret and public keys can be used to encrypt messages using the function *Enc*, where a message can be a vector of elements with s slots or a single element (where $s = 1$). We say that a scheme has SIMD capabilities when $s > 1$. This is the case for example for BGV, B/FV and CKKS. The secret key is also used to decrypt ciphertexts using the function Dec. An HE scheme is correct (a.k.a, "exact") if for every input vector $\bar{m}$ and $0 \leq i < s$, $\bar{m}[i] = \mathrm{Dec}(\mathrm{Enc}(\bar{m}))[i]$, and it is approximately correct (e.g., CKKS) if for some small $\epsilon > 0$ it follows that $|\bar{m}[i] - \mathrm{Dec}(\mathrm{Enc}(\bar{m}))[i]| \leq \epsilon$. The functions Add and Mul are defined for exact schemes as:

$$\mathrm{Dec}(\mathrm{Add}(\mathrm{Enc}(\bar{m}), \mathrm{Enc}(\bar{m}')))[i] = \bar{m}[i] + \bar{m}'[i] \qquad 0 \leq i < s \qquad (1)$$

$$\mathrm{Dec}(\mathrm{Mul}(\mathrm{Enc}(\bar{m}), \mathrm{Enc}(\bar{m}')))[i] = \bar{m}[i] * \bar{m}'[i] \qquad 0 \leq i < s \qquad (2)$$

where when the input is a vector of elements, the function Rot is

$$\mathrm{Dec}(\mathrm{Rot}(\mathrm{Enc}(\bar{m}), n))[i]) = \bar{m}[(i+n) \pmod{s}] \qquad 0 \leq i < s \qquad (3)$$

Approximate schemes use similar definition just with the ϵ notation.

3.2 Experimental setup

Our paper combines several experiments, discussed throughout the different sections to empirically demonstrate the theoretical concepts it presents. For all the experiments, we considered an Intel® Xeon® CPU E5-2699 v4 @ 2.20 GHz machine with 44 cores (88 threads) and 750 GB memory, which we limited to use only 8 cores without hyper-threading (unless otherwise specified). The reason for the above limit is that in CKKS, we store all elements in one ciphertext while in TFHE we store each value in a different ciphertext. As a result, when considering a batch of elements, TFHE's computation scales with the number of CPUs, whereas CKKS depends on the scalability of the underlying bootstrapping implementation. To avoid benchmarking fairness issues that result from idle CPUs, we decided to limit their number to 8. We stress that when using circuits that are more complicated than the serial circuits in some of our experiments, CKKS also scales with the number of CPUs, i.e., it can utilize all cores. Finally, in our experiments we used the CKKS

implementation from the HEaaN library [21]. The HEaaN parameters we used were ciphertexts with 2^{15} coefficients, a multiplication depth of 9, fractional part precision of 24, and integer part precision of 5. This context allows us to use up to 6 multiplications before bootstrapping is required.

3.3 Sign functions

Modern approximate HE schemes only support polynomial operations. Thus, performing branches or comparisons are not trivial and require dedicated utilities. Specifically, they require polynomials that approximately provide a branch functionality such as the step function

$$\mathrm{Step}_\alpha(x) = \begin{cases} 0 & 0 \le x < 0.5 - \frac{\alpha}{2} \\ 1 & 0.5 + \frac{\alpha}{2} < x \le 1 \\ \infty & otherwise \end{cases}, \tag{4}$$

defined for $x \in [0, 1]$ or the sign function

$$\mathrm{Sign}_\alpha(x) = \begin{cases} 0 & -1 \le x < -\alpha \\ 1 & \alpha < x \le 1 \\ \infty & otherwise \end{cases}, \tag{5}$$

where these functions are equivalent because $\mathrm{Sign}_\alpha(x) = 2\,\mathrm{Step}_\alpha(\frac{x+1}{2}) - 1$ for $x \in [-1, 1]$. Thus, we use them interchangeably. We denote the polynomial approximation for these functions by using an additional parameter β. We define $\mathrm{Sign}_{\alpha,\beta}$ and $\mathrm{Step}_{\alpha,\beta}$ such that $|\,\mathrm{Sign}_{\alpha,\beta}(x) - \mathrm{Sign}_\alpha(x)| < \beta$ and $|\,\mathrm{Step}_{\alpha,\beta}(x) - \mathrm{Step}_\alpha(x)| < \beta/2$. Using Step and Sign it is possible to simulate branches and implement comparison functions as well as other primitives such as $\max(a, b)$ and $\min(a, b)$ [16] and the ReLU function as in [34]. We denote by $S(\alpha, \beta)$ and $D(\alpha, \beta)$ the size and depth of the function $\mathrm{Sign}_{\alpha,\beta}$, respectively. Then, Theorem 1 is an existence theorem that is based on [16][Theorem 1] and states that an efficient sign function exists for different α, β values.

Theorem 1 (Variation of [16] Theorem 1) *There exists an efficient sign function $Sign_{\alpha,\beta}(x)$ that for $\alpha < |x| \le 1$ returns y s.t. $|y - Sign(x)| < \beta$ and for which $S(\alpha, \beta) = D(\alpha, \beta) = \mathcal{O}\left(log(1/\alpha)\right) + \mathcal{O}(loglog(1/\beta))$.*

Optimal sign functions.. Several sign approximations were introduced in [16, 33, 34]. All rely on Minimax approximations; they measure the accuracy of the sign function by the maximal error between the target function and its approximating polynomial over a predetermined domain. The sign functions in [16] considers a chain of polynomial composition that involves two functions f_n, g_n i.e., $\mathrm{Sign}(x) \equiv f_n^{d(n)}\left(g_n^{d(n)}(x)\right)$, where n is the logarithm of the

polynomial degree and $d(n)$ is the number of repetitions of the polynomial in the composition. Lee et al. [33] introduced a dynamic algorithm to determine an optimal chain of (different) polynomials $\text{Sign} \equiv p_n\left(p_{n-1}\left(\cdots\left(p_0(x)\right)\right)\right)$ and showed optimal values for the circuit multiplication depth. Based on this work, Lee et.al [34] provide a method for approximating the ReLU and Max function, and showed that they can be replaced in neural networks for high precision inference.

Our work is agnostic to the sign function choice as long as it adheres to the criteria of Theorem 1. It can be as simple as $f_1(x) = -0.5x^3 + 1.5x$ from [16] for $x \in [-1, 1]$, which is equivalent to the step function $h_1(x) = -2x^3 + 3x^2$ for $x \in [0, 1]$, or more complex as $\text{Step}^1(x) = f_3^3(g_3^8(x))$ also from [16] with parameters that satisfy the minimal bounds of [16][Corollary 3] and degree 7 polynomials. For readability, we denote the compare function that is based on Step^1 and compares to elements a, b by $\text{CheckEqual}\,1(a, b)$.

Our work uses the Sign or Step functions to solve a different research question: how to enable high precision computation of a general circuit when the inputs arrive from a discrete set. Specifically, we target generic circuits and not specific function approximation such as ReLU or Max as in [34].

Remark 1 Although BLEACH methodology is agnostic to the choice of sign function, we tried implementing the Remez algorithm and the algorithms from [33, 34] for our integer cleaning method. The coefficients we got when setting $\alpha = 0.1$, $\beta = 0.001$, and $N = 8$ are listed in Appendix A. Although they work fine in plaintext, we could not achieve the required precision under encryption. Thus, we decided to stay with the sign functions of [16], since they achieved good performance and are easy to implement.

3.4 Ciphertext packing using tile tensors

Leveraging the SIMD capabilities of HE schemes such as BGV [12], B/FV [11, 22], and CKKS [15] to speed up the execution of HE circuits often requires the use of complex packing methods. The exact method choice can dramatically affect the *latency* (i.e., time taken to perform the computation), *throughput* (i.e., number of computations performed within a unit of time), communication costs i.e., server-client bandwidth requirement), and memory requirement.

The recent HeLayers [2] framework proposes a data structure called *tile tensor* that packs tensors (*e.g.* vectors, matrices) into fixed-size chunks called *tiles*. Due to their fixed sizes, tile tensors are a natural fit for HE, because each tile can be encrypted into a single ciphertext, where each element of the tile is mapped into a separate slot in the ciphertext.

Tile tensors provide a flexible packing method, where an input tensor can be packed into tiles of different shapes, but of the same size. For instance, while a matrix may be naïvely packed into row or column vectors, it can also be packed into two-dimensional tiles, as long as the tile size matches the number of slots in the ciphertext. The HeLayers framework [2] includes an optimizer that

allows us to navigate the different packing choices in an automatic way, and find the one that best fits the optimization criteria and the user constraints.

An advantage of using a tile tensor is that it supports several types of manipulations, such as duplicating elements along one or more dimensions; this is necessary, for instance, when there is a batch dimension to the input and the data need to be replicated to construct three-dimensional tiles. We further elaborate on how we can use this feature in Section 5.

4 Unbounded binary circuits

Computing unbounded binary circuits over CKKS can be done by treating sign functions as cleanup utilities and combining them with standard binary gates. To this end, we need to a) consider different binary operations; b) transform or approximate them using CKKS operations; c) compute and bound the accumulated error of their results; d) use the cleanup utility to remove the accumulated errors. Following every gate in a binary circuit with an appropriate cleanup function, allows us to evaluate any binary circuit. In practice, it is possible to perform a cleanup after several group of operations and thereby save the number of cleanup invocations. The exact choice depends on the cleanup utility used.

For example, let $x, y \in B_k$ be two binary values and consider the logical-and operation $x \wedge y$, which is implemented in CKKS using one multiplication $r = xy$. According to [1], CKKS loses at most 1 bit of precision per multiplication, thus, $r \in B_{k+1}$. We can now apply $r = \text{Cleanup}_{\left(2^{-(k+1)}, 2^{-k}\right)}(r)$ to return r to B_k.

Table 2: A comparison of running a binary circuit with 2^{15} elements over CKKS compared to TFHE. See more details in the text.

	CKKS			**TFHE**	
Binary operation	Approximation	Latency per iter. (sec)	Amortized latency per iter. (μsec)	Latency per iter. (sec)	Amortized latency per iter. (μsec)
$x \wedge y$	$x \cdot y$	2.26	69	440	13,420
$x \vee y$	$x + y - x \cdot y$	2.3	70	440	13,420
$x \oplus y$	$(x - y)^2$	2.4	73	440	13,420

In what follows, we show that bit operations (in Table 2) followed by calling h_1 can lead to fully-HE, depending on the parameters of the key. We demonstrate this by proving that throughout the circuit, we can keep an invariant in which the noise is smaller than some constant. We first prove that the error does not grow by much after one binary operation. Then we show that for small errors the h_1 decreases the error by a large factor. Then we conclude by demonstrating that the noise added by CKKS, for appropriate parameters of the key, is small enough to keep the error small.

We start by bounding the amount of error added by a binary operation.

Lemma 2 *For $x, y \in B_2$, i.e., $x = b_x + e_x$ and $y = b_y + e_y$, where $b_x, b_y \in \{0, 1\}$ and $|e_x|, |e_y| < e < 0.25$. Then*

$$|(x \wedge y) - (b_x \wedge b_y)| < 5e,$$
$$|(x \vee y) - (b_x \vee b_y)| < 5e,$$
$$|(x \oplus y) - (b_x \oplus b_y)| < 5e.$$

Proof

And.

$$\begin{aligned} |(x \wedge y) - (b_x \wedge b_y)| &< |b_x \cdot b_y + e_y b_x + e_x b_y + e_x e_y - b_x \cdot b_y| \\ &< |2e + e^2| < 5e. \end{aligned} \tag{6}$$

Or.

$$\begin{aligned} |(x \vee y) - (b_x \vee b_y)| &= |(1-x)(y-1) + 1 - ((1-b_x)(b_y - 1) + 1)\,| \\ &= |-1| \cdot |(1-x) \wedge (1-y) - ((1-b_x) \wedge (1-b_y))| \\ &< |2e + e^2| < 5e. \end{aligned} \tag{7}$$

Xor.

$$\begin{aligned} |(x \oplus y) - (b_x \oplus b_y)| &= |(x-y)^2 - (b_x - b_y)^2| \\ &= |(x - b_x - (y - b_y))(x + b_x - y - b_y)| \\ &< |2e \cdot (2 + 2e)| = 4e + 4e^2 < 5e. \end{aligned} \tag{8}$$

We now observe that for a small enough region around 0 and 1, the h_1 function reduces the error significantly.

Observation 2.1 If $x = b_x + e_x$, where $b_x \in \{0, 1\}$ and $|e_x| < 0.007$, then $h_1(x) = b_x + e'_x$, where $|e'_x| < 0.1|e_x|$

The proof is algebraic and we leave the details out. The main idea is to use the mean value theorem, using the fact that the derivatives $h'_1(1) = h'_1(0) = 0$. We now observe that the error added during the CKKS rescale can be made small by properly setting the parameters of the key.

Observation 2.2 For proper parameters of the encryption key, the error added when multiplying and rescaling two ciphertexts is smaller than $e_{ckks} = 0.0007$.

Finally, we conclude that if two inputs of a binary operation have a small error, then applying h_1 on the output also yields a small noise and thus maintains the invariant's a small noise.

Lemma 3 *Let $x = b_x + e_x$ and $y = b_y + e_y$ be input to a binary operation, $b_x, b_y \in \{0, 1\}$ and $|e_x|, |e_y| < e \leq 0.0001$, and the error added when multiplying and rescaling two ciphertexts is e_{ckks} such that $2.1e_{ckks} < 0.5e$. Then $z = b_z + e_z$, where $b_z \in \{0, 1\}$ and $|e_z| < e$ for:*

$$z = h_1(x \wedge y) \text{ or}$$
$$z = h_1(x \vee y) \text{ or}$$
$$z = h_1(x \oplus y)$$

Proof Let $z' = x$ *op* y (where *op* is $\wedge, \vee$ or $\oplus$). Then $z' = b_z + e_{z'}$, where $|e_{z'}| < 5e + e_{ckks}$. Since $5e + e_{ckks} < 6e < 0.007$ we have: $h_1(b_z + 5e + e_{ckks}) < b_z + 0.1(5e + e_{ckks})$. Applying h_1 requires two rescale operations and so $|e_z| < 0.5e + 2.1e_{ckks} < e$.

4.1 Binary operations experiments

To show the efficiency of our binary circuits approach, we compare the performance of running the binary AND, OR, and XOR operations over CKKS with their performance over TFHE. Note that the NOT unary operation translates in CKKS to $\text{NOT}(x) = 1 - x$, which does not add significant noise and therefore does not require cleaning.

As explained above, the advantage of using CKKS comes from its SIMD capability. We leveraged this capability by setting the number of parallel gates equal to the number of slots in the experimented ciphertext. Specifically, we used ciphertexts that can accommodate $s = 2^{15}$ elements at once. As stated previously, our goal is not to show that CKKS is *always* better than other schemes such as TFHE, rather to show that in some cases it is better to use it even for binary circuits.

In our experiment, we set the cleanup utility to h_1, and set the circuit depth to $d = 100$. We repeatedly called the same gate d times and performed a cleanup operation after every such call. We called a bootstrapping operation after every two gate+Cleanup executions. To account for the latency of the bootstrapping operations, we included it when averaging the latency per gate measures. Table 2 summarizes our experiments. It shows the operations, their equivalent CKKS approximation functions, the overall latency of running the same gate d times divided by d, and the amortized latency of the computations when dividing the latency with the number of slots (parallel elements).

The inputs to the AND and OR operators were a random vector x, which operated on itself. This keeps the balance between slots with values of 0 and 1 after every iteration. The input to the XOR operation was a vector x with values chosen uniformly at random and another such vector y that was generated at every iteration. While the input values do not affect the latency, they allowed us to verify that the decrypted output and the expected output after d gates is the same.

To measure the latency and amortized latency of performing the different Boolean gates in TFHE, we used Concrete version 0.2.0-beta [19] and wrote a Rust code that follows the above experiment. The code generated two FheBool

vectors and performed element-wise bit operations to get the final results. To leverage the multi-core environment we used Rayon's `into_par_iter`, which automatically splits the work among the different running threads. To avoid race conditions on the `ServerKey` object we cloned it several times and locked an instance per thread as demonstrated in [23]. The results matched the expected results of ∼13 milliseconds per operation and were scaled linearly when modifying the cores number between 1-8. As can be seen from Table 2, the amortized latency speedup observed in our experiment is around two order of magnitudes in favor of the CKKS implementation.

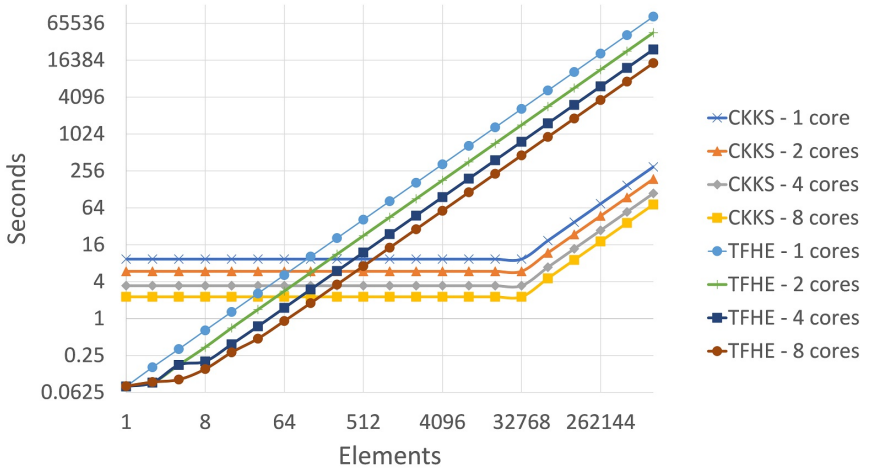


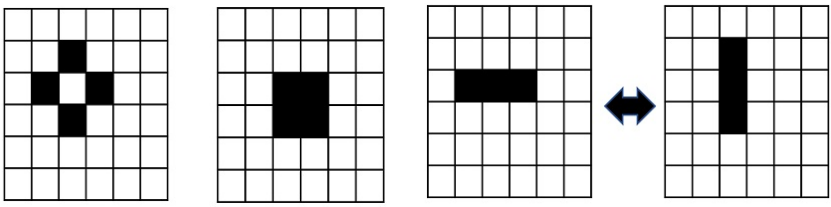
Figure 3: A comparison of TFHE and CKKS latency when performing m (X-axis) AND gates in parallel using $1/2/4/8$ cores. For $m > 1{,}024$ data was extrapolated. The X-axis and Y-axis are in logarithmic scale.

To get a better feeling of our comparison experiment, Figure 3 extrapolates the data from Table 2 to a different number m of similar gates running in parallel. In addition, it presents the results when setting the number of cores to $1, 2, 4$, and 8. Note that while the graphs look linear the axes are logarithmic, we see that for small values of $m = 64, 128, 256, 512$ it is better to run TFHE over CKKS for $1, 2, 4, 8$ cores, respectively. For larger values of m, CKKS is better.

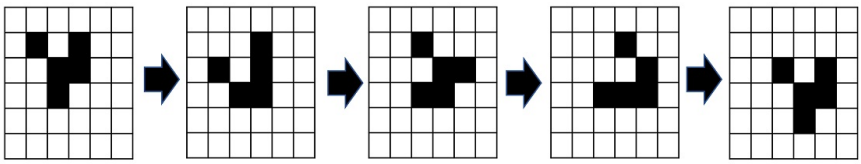
The CKKS ciphertexts involved 2^{15} slots. Therefore, the latency was always the same for $m < 2^{15}$. Also, the number of cores did not affect the results because only one ciphertext was involved in the computations. Still, 8 CPU cores were used due to the parallelization of the bootstrapping operations. When using more than 8 cores, we observed idle CPU cores for CKKS.

5 Game of Life

We already saw that it is possible to achieve better performance for certain binary circuits when using CKKS compared to non-SIMD capable schemes such as TFHE. In this section, we consider a more complex circuit that involves binary and integer operations to demonstrate the cleanup technique, namely Conway's Game of Life [24]. We chose this demonstration as it simulates a deep computation that can be executed as deep as required. We also selected it for the reasons specified in [42, 43], where the Game of Life was implemented and executed over TFHE. While the outcome of this iterative game is deterministic, it is hard to predict its results for a given iteration without running all previous iterations. An interesting property of this game is that it is Turing complete [48], which means the game can theoretically simulate every sequence of computer operations.



(a) Stable state 1. (b) Stable state 2. (c) An oscillator state with period 2.



(d) A glider state with period 4.

Figure 4: Several famous examples of Game of Life instances of size 6×6. Black cells are live cells.

Rules. The Game of Life is an iterative process that gets a starting bit-string in the form of a grid, which "evolves" at every iteration. The game does not limit the size of the grid up to the available memory. Here, as in [42, 43] we bounded the grid dimensions and treated it as an $n \times n$ matrix that holds a bit indicator per cell, indicating whether it is "dead" or "alive". At every iteration, a cell is either "born", "survives", or "dies", depending on the state of its 8 neighbors. A dead cell is born when it has exactly 3 live neighbors and stays dead otherwise. A live cell survives if it has exactly 2 or 3 live neighbors and it dies otherwise. Figure 4 presents several common examples. Panels (a)

and (b) are stable states in which all cells remain in the same state at every iteration. Panel (c) shows an oscillator that only has two states: one for odd iterations and another for even iterations. Panel (d) demonstrates a period 4 glider, which is a periodic form that appears one row below and one column to the right after every 4 iterations.

Algorithm 1 describes 1 iteration of the Game of Life. It receives an $N \times M$ board B as input, performs 1 iteration and outputs the updated board B'. It contains 2 main operations: a) counting the live neighbours of every cell in B at Steps 3-6; b) evaluating the cell state (dead or alive) in Step 7. For the latter operation it uses the CheckEqual(a, b) function, which returns an indicator for whether $a = b$.

Algorithm 1 ComputeGoLStep

input: $N, M \in \mathbb{N}$ and B an $M \times N$ Game of Life board.
output: B' an updated board.

```
1:  procedure ComputeGoLStep(B)
2:      for every cell (i, j) in B do
3:          n := 0
4:          for every neighbour cell (x, y) of the cell (i, j) do
5:              n+ = B[x][y]
6:          end for
7:          B'[i][j] := CheckEqual(n, 2) · B[i][j] + CheckEqual(n, 3)
8:      end for
9:      return B'
10: end procedure
```

5.1 Implementation

We implemented the Game of Life using CKKS and the interleaved tile tensors packing method of [2], see Section 3.4. Figure 5 provides an example packing a 6×6 Boolean board into 9 ciphertexts when using tile tensors of shape $\left[\frac{6^-}{2}, \frac{6^-}{2}\right]$. Here, every 9 items (blue dashed line) are spread among the tile-tensor internal tiles in an interleaved way. The colors of every input item indicate its destination ciphertext. An alternative packing option was to place every cell of the game in a different ciphertext. While easy to understand, this solution was not very efficient compared to the above, which allowed us to drastically reduce the number of ciphertexts, run-time, and memory consumption. In addition, the use of HeLayers [2] allowed us to pack the data using a simple one line API.

To perform the neighbor counting method, we used the HeLayers SumPooling API, which executes a 3×3 convolutional filter with all cells set to 1. Subsequently, we reduced the value of the inspected cell (i, j) from the results.

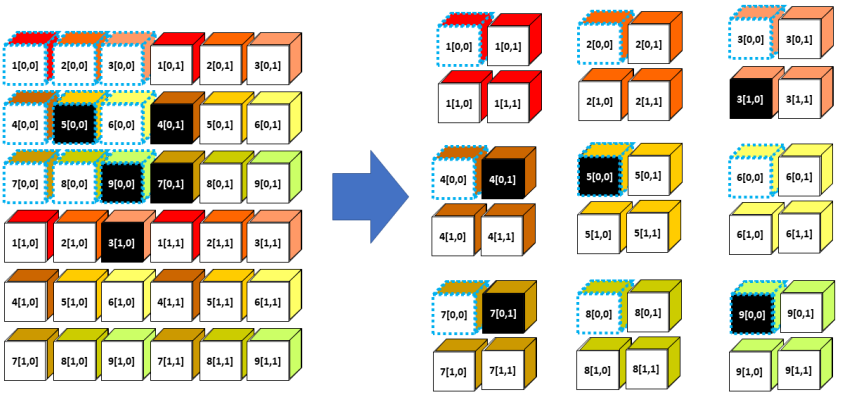


Figure 5: An example of interleaved tile tensor packing of the glider from Figure 4. Packing a 6×6 matrix to 9 different ciphertexts. The tile tensor shape is $\left[\frac{6\tilde{}}{2}, \frac{6\tilde{}}{2}\right]$.

Evaluating new board states. Algorithm 1 invokes the comparison function CheckEqual(c, p) at Line 7 twice with $p = 2$ and $p = 3$. It does so to determine whether a cell should live or die. This function receives a ciphertext c and a plaintext p, and returns 1 when $\text{Dec}(c) \approx p$ and 0 otherwise. To implement it, we explored two approaches: i) using CheckEqual 1; ii) using Lagrange interpolation.

For Game of Life, an implementation of CheckEqual should only consider $p = 0, \ldots, 8$, which is the number of neighbors computed in Algorithm 1, Lines 3-6. Our Lagrange interpolation implementation of CheckEqual leverages the above property. It is defined using the following degree 8 polynomials

$$F_p(x) = \prod_{i \in \{0,\ldots,8\}/\{p\}} \frac{(x-i)}{(p-i)} \tag{9}$$

where the denominator is fixed and can be pre-computed. Also here, because we deal with binary boards we applied the h_1 Cleanup utility on the binary cells. To get the cleanup function parameters, we used the upper bound of 1 bit of noise per multiplication and took into consideration the scaling done by the Lagrange polynomial. This ensured us that applying h_1 Cleanup after every operation will keep the noise level.

Reference implementation. For the reference implementation we used the code of [52], which uses Concrete [19] and `FheBool` elements. We follow the guidelines on [42] to modify the code so it also uses `FheUint2` and `FheUint3`. Finally, we used the parallelization technique presented in [23]. Specifically, to

ensure thread-safe code, we needed to clone the server keys and the board state before calling the `update` function. The cloning procedure was not included in our latency measurements.

5.2 Experiments

Performance. We ran our implementation on boards of size 128×128 and 256×256. For every board, we played the game for a large number of iterations (more than 50) and computed the average time per iteration. The average latency when using the general compare method CheckEqual 1 was 22:12 and 39:40, respectively. The average latency when considering our custom Lagrange interpolation was 6:08 and 12:00 minutes, respectively, 3.619× and 3.3× faster. In both cases, we see a linear growth while the board size increased quadratically.

Figure 6 compares our CKKS-based results with the reference TFHE implementation. It can be observed that for boards of size larger than around 80×80, CKKS performed an order of magnitude faster than the TFHE implementations. Surprisingly, the TFHE implementation that used $FheUint2$ elements turned out to be the fastest among the TFHE implementations.

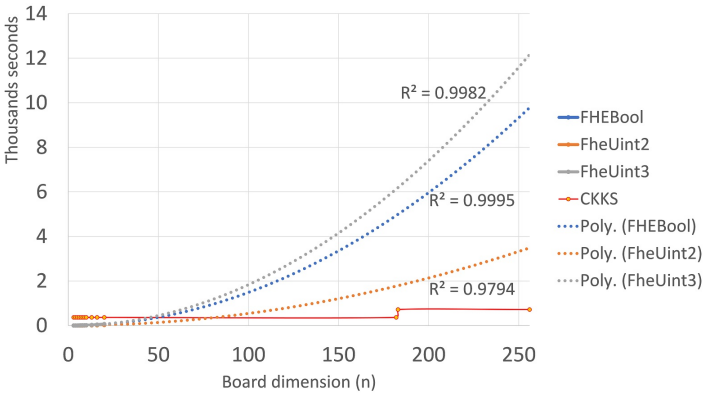
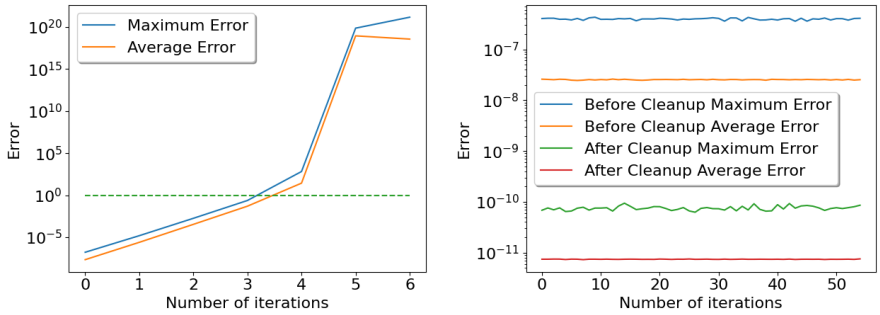


Figure 6: A comparison of our CKKS implementation with the reference TFHE implementations, where lower values are better. The X-axis is the board dimension n, i.e., the board size is $n \times n$. For TFHE we only measured boards of $n \leq 20$ and extrapolate the results to larger boards using quadratic extrapolation.

Accuracy. Our naïve CKKS implementation that did not use BLEACH went out of sync after just a few iterations, as shown in Figure 7 Panel (a). In contrast, our implementation with the cleanup utility showed the same results as a plaintext version of the game using the same initial state of cells, as shown in Figure 7 Panel (b). Here, we observe that the error is maintained at a constant level after every iteration.



(a) No cleanup is done.

(b) A cleanup after every iteration.

Figure 7: Average (orange line) and maximum (blue line) error (y-axis) observed after the ith iteration (x-axis) on the cells of a 128×128 board. Panel (a) shows that without cleanup, the error explodes (crosses the horizontal bar at y=1) after 4 iterations. In contrast, Panel (b) shows the error before (blue and orange lines) and after (green and red lines) every iteration. Even after 50 iterations the error is stable. Also note how the error before the cleanup is higher than the error after the cleanup (both the average and the maximum error). This proves the "cleaning" property of BLEACH.

6 Bit decomposition

Previous examples considered binary circuits and circuits of binary inputs using integer operations. Here, we take the cleanup utility one step further, and present an algorithm that decomposes a number x to its binary representation under FHE. For simplicity, we assume $x \in \mathbb{N}$ and $x < 2^{N+1}$ for some predefined small N. Fixed precision numbers can be dealt with using scaling. In a nutshell, our bit-decomposition algorithm works by extracting the most significant bit (MSB) of x and then setting $x' = x - 2^N \cdot (x)_N$ and continuing with a new upper bound $x' < 2^N = 2^{N'+1}$.

The MSB extraction algorithm is described in Algorithm 2. The algorithm is defined according to the parameters $N \in \mathbb{N}$ and $\alpha, \beta, e_{ckks} \in \mathbb{R}_{\geq 0}$, where e_{ckks} is an upper bound on the error that may be added during the execution of the algorithm. This bound depends amongst others on the parameters of the key, the implementation of the sign function, and the implementation of bootstrapping (if needed). The algorithm gets an input $[\![x + e]\!]$ such that $e \in \mathbb{R}$ is a small number $|e| < 0.5 - \alpha$. It outputs an encrypted bit $[\![b]\!]$ which is a β approximation of $(x)_N$, i.e., $|b - (x)_N| < \beta$.

The following lemma considers the correctness and complexity of Algorithm 2 and is used by Theorem 5.

Lemma 4 *Let α, x, e, N be parameters as in Algorithm 2, then*

$$\frac{\alpha}{2^N} < \left| \frac{x + e + 0.5}{2^N} - 1 \right| < 1.$$

Algorithm 2 ExtractMSB$_{N,\alpha,\beta,e_{ckks}}$

parameters: $N \in \mathbb{N}$, $\alpha, \beta, e_{ckks} \in \mathbb{R}_{\geq 0}$.
input: A ciphertext $[\![x+e]\!]$, where $x \in \mathbb{N}$, $x < 2^{N+1}$, and $|e| < 0.5 - \alpha$.
output: A ciphertext $[\![b]\!]$ s.t. $|b - (x)_N| < \beta$.

1: **procedure** ExtractMSB$_{N,\alpha,\beta,e_{ckks}}([\![x+e]\!])$
2: $\quad [\![x']\!] := \left[([\![x+e]\!] + 0.5)/2^N\right] - 1$
3: $\quad \alpha' := \frac{\alpha}{2^N}$ ▷ Can be pre-computed
4: $\quad \beta' := 2(\beta - e_{ckks})$ ▷ Can be pre-computed
5: $\quad [\![b']\!] := \text{Sign}_{\alpha',\beta'}(x')$
6: $\quad [\![b]\!] := ([\![b']\!] + 1)/2$
7: $\quad$ **return** $[\![b]\!]$
8: **end procedure**

Proof Denote $x_{fl} = x + e + 0.5$ and $x' = \frac{x_{fl}}{2^N} - 1$ as in Algorithm 2 Line 2. From the parameters of Algorithm 2 we get $|e| < 0.5 - \alpha$ so

$$x - 0.5 + \alpha < x + e < x + 0.5 - \alpha \tag{10}$$

and

$$x < x + \alpha < x_{fl} < x + 1 - \alpha < x + 1 \tag{11}$$

when $0 \leq x < 2^{N+1}$ we distinguish between 2 cases

Case 1	Case 2
$0 \leq x < 2^N$	$2^N \leq x < 2^{N+1}$
$0 < x_{fl} < 2^N - \alpha$	$2^N + \alpha < x_{fl} < 2^{N+1}$
$-1 < x' < -\frac{\alpha}{2^N}$	$\frac{\alpha}{2^N} < x' < 1$

where in both cases $\frac{\alpha}{2N} < |x'| < 1$ as required.

Theorem 5 *Let $\alpha, \beta, x, e, N, e_{ckks}$ be parameters as in Algorithm 2 then*

1. *Algorithm 2 is correct, i.e., it outputs b, where $|b - (x)_N| < \beta$*
2. *Algorithm 2 requires a circuit of size $O(S(\frac{\alpha}{2^N}, 2\beta))$ and depth $O(D(\frac{\alpha}{2^N}, 2\beta))$.*

Proof

1. Let $x' = \frac{x+e+0.5}{2^N} - 1$, $\alpha' = \frac{\alpha}{2N}$ and $\beta' = 2(\beta - e_{ckks})$ as in Algorithm 2, Lines 2, 3, and 4, respectively. From Lemma 4 we know that $\alpha' < |x'| < 1$ so that by Thm. 1 when we apply $b' = \text{Sign}_{\alpha',\beta'}(x')$ on Line 5 we get that $|b' - (x)_N| < \beta'$. Subsequently, because $(x)_N \in \{-1, 1\}$ we get $b' = (2(x)_N - 1) + e_1$, where $|e_1| < \beta'$ i.e., $\frac{|e_1|}{2} + e_{ckks} < \beta$. It follows that $b = (x)_N + e_2$, where $|e_2| \leq |\frac{e_1}{2} + e_{ckks}| \leq \frac{|e_1|}{2} + e_{ckks} \leq \beta$.

Algorithm 3 $\text{Decompose}_{N,\alpha,\beta}$

parameters: $N \in \mathbb{N}$, $\alpha, \beta, e_{ckks} \in \mathbb{R}_{\geq 0}$.
input: A ciphertext $[\![x+e]\!]$, where $x \in \mathbb{N}$, $x < 2^{N+1}$, and $|e| < 0.5 - \alpha$.
output: A vector $([\![b_i]\!])_{0 \leq i < N}$ of the $N+1$ bits of x s.t. $|b_i - (x)_i| < \beta_i$.

1: **procedure** $\text{Decompose}_{N,\alpha,\beta_N,\ldots,\beta_0}([\![x+e]\!])$
2: $\quad [\![y_N]\!] := [\![x]\!]$
3: $\quad$ **for** $i := N, \ldots, 0$ **do**
4: $\quad\quad \alpha_i := \alpha \cdot 2^{i-N}$
5: $\quad\quad \beta_i := \min(\beta, \alpha \cdot 2^{i-N})$
6: $\quad\quad [\![b_i]\!] := \text{ExtractMSB}_{i,\alpha_i,\beta'_i,e_{ckks}}([\![y_i]\!])$
7: $\quad\quad [\![y_{i-1}]\!] := [\![y_i]\!] - 2^i \cdot [\![b_i]\!]$
8: $\quad$ **end for**
9: $\quad$ **return** $([\![b_i]\!])_{0 \leq i < N}$
10: **end procedure**

2. This follows from Thoerem 1, since Algorithm 2 performs exactly one call to $\text{Sign}_{\frac{\alpha}{2^N}, 2\beta}$ together with a fixed number of plaintext-ciphertext multiplications.

We now move to describe our bit decomposition algorithm. As before we get an input $x + e$, where $x \in \mathbb{N}$ and $|e| < 0.5 - \alpha$. Algorithm 3 uses Algorithm 2 iteratively to find the MSB and remove it from the input x. Attention should be given to the parameters of Algorithm 2. Specifically, α needs to be adjusted to account for the error coming from subtracting approximated bits in previous iterations.

Lemma 6 *Let $N, \alpha, \beta_N, \ldots, \beta_0, x, e$ be parameters as in Algorithm 3 and $y_0, \ldots y_N$ as defined by Algorithm 3 Lines 2 and 7. Then $|y_i - (x \mod 2^{i+1})| < 0.5 - \alpha 2^{i-N}$, for $0 \leq i \leq N$.*

Proof We prove this by induction. Starting from the first iteration ($i = N$) and going to the Nth iteration ($i = 0$). The case $i = N$ follows immediately from the assumptions on the input

$$|y_N - x| = |x + e - x| = |e| < 0.5 - \alpha \cdot 2^{N-N} = 0.5 - \alpha. \tag{12}$$

Assume it holds that

$$|y_i - (x \mod 2^{i+1})| < 0.5 - \alpha \cdot 2^{i-N}, \tag{13}$$

then we prove that

$$|y_{i-1} - (x \mod 2^i)| < 0.5 - \alpha \cdot 2^{i-1-N}. \tag{14}$$

Starting from

$$|y_{i-1} - (x \mod 2^{i-1})|$$

$$
\begin{aligned}
&= |y_{i-1} - y_i + y_i - (x \mod 2^{i-1}) + (x \mod 2^i) - (x \mod 2^i)| \\
&\leq |y_{i-1} - y_i + (x \mod 2^i) - (x \mod 2^{i-1})| + |y_i - (x \mod 2^i)| \\
&< |y_{i-1} - y_i + (x \mod 2^i) - (x \mod 2^{i-1})| + 0.5 - \alpha \cdot 2^{i-N}, \qquad (15)
\end{aligned}
$$

where the last inequality follows from the induction assumption on Equation (13). By construction (Alg. 3 Line 7) we have $y_{i-1} - y_i = 2^i \cdot b_i$, where b_i is a ciphertext and multiplying by 2^i does not add noise because it can be implemented by i additions of elements with low error. Since $(x \mod 2^i) - (x \mod 2^{i-1}) = 2^i \cdot (x)_i$ we get

$$(15) \leq |2^i \cdot (b_i - (x)_i)| + 0.5 - \alpha \cdot 2^{i-N} \qquad (16)$$

By construction (Alg. 3 Line 5)

$$|b_i - (x)_i| \leq \beta_i \leq \alpha 2^{-N-1} \qquad (17)$$

and so

$$(16) \leq 2^i \cdot \alpha \cdot 2^{-N-1} + 0.5 - \alpha \cdot 2^{i-N} \qquad (18)$$

$$= 0.5 - \alpha(2^{i-N} - 2^{i-N-1}) = 0.5 - \alpha 2^{i-N-1}. \qquad (19)$$

as required.

Theorem 7 *Let N, α, β, e be parameters as in Alg. 3, then*

1. *Algorithm 3 outputs $([\![b_i]\!])_{0 \leq i < N}$, s.t. $|b_i - (x)_i| < \beta_i$.*
2. *Algorithm. 3 requires a circuit of size $O(\sum S(\alpha \cdot 2^{i-N}, \min(\beta, \alpha \cdot 2^{i-N})))$ and depth $O(\sum D(\alpha \cdot 2^{i-N}, \min(\beta, \alpha \cdot 2^{i-N})))$, where $S(\alpha, \beta)$ and $D(\alpha, \beta)$ are the size and depth of the circuits implementing the sign function with parameters α and β.*

Proof

1. The ith bit b_i is computed in Line 6 by calling $\text{ExtractMSB}_{i,\alpha_i,\beta_i,e_{ckks}}(y_i)$. From Lemma 6 we have that $|y_i - x| < \alpha_i$ and $2^{i+1} > y_i \in \mathbb{N}$ as required from the parameters and input of ExtractMSB.
2. Algorithm 3 involves N calls to ExtractMSB with a total circuit size of $O(\sum S(\alpha \cdot 2^{i-N}, \min(\beta, \alpha \cdot 2^{i-N})))$ and circuit depth $O(\sum D(\alpha \cdot 2^{i-N}, \min(\beta, \alpha \cdot 2^{i-N})))$, where $S(\alpha, \beta)$ and $D(\alpha, \beta)$ are the size and depth of the circuits implementing the sign function with parameters α and β.

Corollary 7.1 Let $N \in \mathbb{N}, \alpha, \beta \in \mathbb{R}_{\geq 0}$ be parameters as in Algorithm 4, then Algorithm 4 requires a circuit of size $O(N \cdot \log(\frac{1}{\alpha}) + N^2 + N \cdot \log(\log(\frac{1}{\beta})))$ and depth $O(N \cdot \log(\frac{1}{\alpha}) + N^2 + N \cdot \log(\log(\frac{1}{\beta})))$.

Proof Using [16] Theorem 1, we can achieve sign function of parameters α, β by circuits of size and depth $O(-\log(\alpha) + \log(-\log(\beta)))$. Using this, with Theorem 8, we get the size and depth of the circuit required by Algorithm 4 are

$$O(\sum -\log(\alpha \cdot 2^{i-N}) + \log(-\log(\min(\beta, \alpha \cdot 2^{i-N}))))$$

$$
\begin{aligned}
&= O(\sum -\log(\alpha \cdot 2^{i-N}) + \log(-\log(\beta))) \\
&= O(\sum -\log(\alpha \cdot 2^{i-N}) + \log(-\log(\beta))) \\
&= O(\sum (-\log(\alpha) - i + N + \log(\log(\frac{1}{\beta})))) \\
&= O(N \cdot \log(\frac{1}{\alpha}) + N^2 + N \cdot \log(\log(\frac{1}{\beta})))
\end{aligned}
$$

7 Integer Computation

We describe an algorithm for "cleaning integers" (Algorithm 4). The input is a message $m_{in} = x + e$, where $x \in \mathbb{N}, x < 2^N$ and $|e| < 0.5 - \alpha$ for some $N \in \mathbb{N}$ and the output is m_{out}, which is a cleaner version of m_{in}, $|m_{out} - x| < \beta < \alpha$, for parameters $\alpha, \beta \in \mathbb{R}_{\geq 0}$. Algorithm 4 calls Alg. 3 to extract the bit decomposition $(\llbracket b_i \rrbracket)_{0 \leq i < N}$ of x at Line 3 and re-compose the cleaned integer at Line 4.

Theorem 8 *Let $N, \alpha, \beta, e_{ckks}$ be parameters as in Algorithm 4 and $\llbracket x + e \rrbracket$ be the input to the protocol, where $x \in \mathbb{N}$, $x < 2^{N+1}$ and $|e| < 0.5 - \alpha$, then*

1. *Algorithm 4 is correct, i.e., it outputs y, s.t. $|y - x| < \beta$.*
2. *Algorithm 4 requires a circuit of size $O(\sum S(\alpha \cdot 2^{i-N}, \min(\beta \cdot 2^{-N-1}, \alpha \cdot 2^{i-N})))$ and depth $O(\sum D(\alpha \cdot 2^{i-N}, \min(\beta \cdot 2^{-N-1}, \alpha \cdot 2^{i-N})))$.*

Proof

1. In Line 3 Algorithm 4 approximately decomposes $x + e$ by calling $\text{Decompose}_{N,\alpha,\beta',e_{ckks}}(\llbracket x + e \rrbracket)$, with $\beta' = 2^{-N-1} \cdot \beta$ that returns $(\llbracket b_i \rrbracket)_{0 \leq i \leq N}$. Next, on Line 4, it computes $y = \sum 2^i b_i$ and we have

$$|y - x| = |\sum 2^i b_i - x| = |\sum 2^i b_i - \sum 2^i (x)_i| < \sum 2^i |b_i - (x)_i| \quad (20)$$

   From Theorem 7, $|b_i - (x)_i| < \beta'$ so that

$$(20) < \sum 2^i \beta' < \sum 2^i 2^{-N-1} \beta < \beta. \quad (21)$$

2. The protocol involves $N + 1$ calls to $extractMsb$ with a total circuit size of $O(\sum S(\alpha \cdot 2^{i-N}, \min(\beta \cdot 2^{-N-1}, \alpha \cdot 2^{i-N})))$ and circuit depth $O(\sum D(\alpha \cdot 2^{i-N}, \min(\beta \cdot 2^{-N-1}, \alpha \cdot 2^{i-N})))$, where $S(\alpha, \beta)$ and $D(\alpha, \beta)$ are the size and depth of the circuits implementing the sign function with parameters α and β.

Corollary 8.1 Let $N \in \mathbb{N}$, and $\alpha, \beta \in \mathbb{R}_{\geq 0}$ be parameters as in Algorithm 4, then Algorithm 4 requires a circuit of size $O(N \cdot \log(\frac{1}{\alpha}) + N^2 + N \cdot \log(N + 1 + \log(\frac{1}{\beta})))$ and depth $O(N \cdot \log(\frac{1}{\alpha}) + N^2 + N \cdot \log(N + 1 + \log(\frac{1}{\beta})))$.

Algorithm 4 CleanInteger$_{N,\alpha,\beta}$

parameters: $N \in \mathbb{N}$, $\alpha, \beta, e_{ckks} \in \mathbb{R}_{\geq 0}$.
input: A ciphertext $[\![x+e]\!]$, where $x \in \mathbb{N}$, $x < 2^{N+1}$, and $|e| < 0.5 - \alpha$.
output: A ciphertext $[\![y]\!]$ s.t. $|y - x| < \beta$.

1: **procedure** CleanInteger$_{N,\alpha,\beta}([\![x+e]\!])$
2: $\quad \beta' := 2^{-N-1} \cdot \beta$
3: $\quad ([\![b_i]\!])_{0 \leq i \leq N} := \text{Decompose}_{N,\alpha,\beta',e_{ckks}}$
4: $\quad [\![y]\!] := \sum [\![b_i]\!] \cdot 2^i$
5: $\quad$ **return** $[\![y]\!]$
6: **end procedure**

Proof Using [16][Theorem 1], we can achieve a $\text{Sign}_{\alpha,\beta}$ function by circuits of size and depth $O(-\log(\alpha) + \log(-\log(\beta)))$. Plugging it in Theorem 8 and denoting $\gamma = \sum -\log(\alpha \cdot 2^{i-N})$ we get that the size and depth of Algorithm 4 circuit are

$$\begin{aligned} & O(\gamma) + \log(-\log(\min(\beta \cdot 2^{-N-1}, \alpha \cdot 2^{i-N})))) \\ & = O(\gamma + \log(-\log(\beta \cdot 2^{-N-1}))) \\ & = O(\sum(-\log(\alpha) - i + N + \log(N + 1 + \log(\frac{1}{\beta})))) \\ & = O(N \cdot \log(\frac{1}{\alpha}) + N^2 + N \cdot \log(N + 1 + \log(\frac{1}{\beta}))) \end{aligned}$$

Corollary 8.2 Let Algorithm 4 for parameters α, β use $\text{Sign}_{\alpha',\beta'} = f_3$ from [16] and let the error B generated by $\text{Sign}_{\alpha',\beta'}$ be less than 0.027855. Then, Algorithm 4 is correct if

1. $\alpha \cdot 2^{-N} < 2.066B$
2. $\min(\beta \cdot 2^{-N-1}, \alpha \cdot 2^{-N}) > 9B$.

Proof Let α'_i, β'_i be as defined in Algorithm 3 Lines 3 and 4, respectively, during the ith iteration. Setting $n = 3$ in [16][Theorem 6] leads to $c_3 = \frac{35}{16}$, $B < 0.0279$ (for appropriate choice of the parameters of the key) and effective bounds on α'_i, β'_i. We now tie these bounds to α, and β. From [16][Theorem 6] $\max_i \alpha'_i = \alpha \cdot 2^{-N} < (\frac{c_3}{c_3 - 1})^{c_3 - 1} B \approx 2.066B$. In addition, $\max_i \beta'_i = -\log(\min(\beta \cdot 2^{-N-1}, \alpha \cdot 2^{i-N})) < log(\frac{1}{B}) - log(9) = log(1/9B)$.

7.1 Experiments

Algorithm 4 describes our integers' cleanup utility, and Theorem 8 showed its correctness. It is left to argue that this method is efficient for some values of N and thus can be manifested in cryptographic libraries that implement CKKS. To this end, we designed the following experiment.

Experiment design. We sample 10,000 random 16-bit integers, where an error, sampled uniformly from $[-0.1, +0.1]$ is added to every integer. The noisy

integers are packed and encrypted as a single CKKS ciphertext that is fed as an input to the cleanup algorithm with the parameters $\alpha = 0.1$, $\beta = 10^{-3}$, and $N = 16$. With these parameters, the algorithm expects a 16-bit integer that contains a maximum error of $|0.1|$, and the expected error of the result should not exceed $|10^{-3}|$. Next, the resulting ciphertext is decrypted and the error level is evaluated by subtracting the decrypted result from the originally sampled integers.

Experiment implementation. We implemented the Algorithm 2 using the Sign polynomial approximation of [16] with parameters that satisfy the minimal bounds of [16][Corollary 3] and degree 7 polynomials.

Experiment results. Our experiment results are presented in Figures 8, 9, and 10. Figure 8 shows a histogram of the measured error after decrypting the results of Algorithm 4. As expected, all the observed errors are positioned to the left of the target error $\beta = 10^{-3}$. In fact, they are orders of magnitude smaller than β. This is expected, since we computed the bounds for the worst case, as can be shown in the proof of theorem 8.

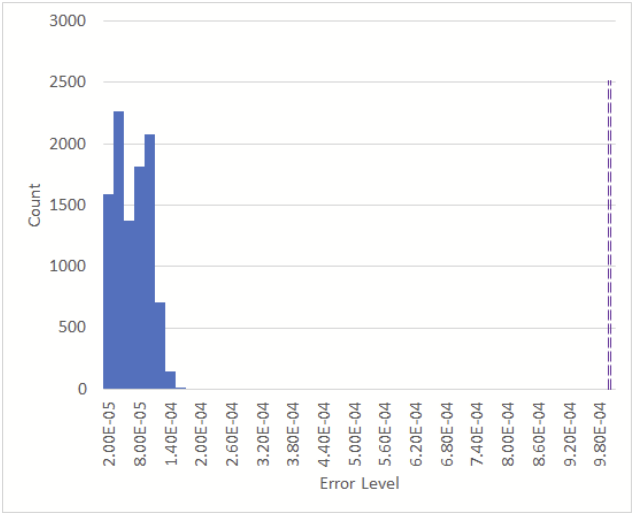


Figure 8: A histogram of the measured error after decrypting the results of $\text{CleanInteger}_{N=16,\alpha=0.1,\beta=10^{-3}}(\cdot)$. All the observed errors are order of magnitude smaller then the requested bound β.

Now that we empirically saw the correctness of our algorithm, we present its latency per bit and the number of required multiplication per bit in Figure 9. It took around 30 seconds to clean up the first bit and 80 seconds to clean up the last bit. The number of performed multiplications was proportional to the latency e.g., 36 and 79 for the first and last bit, respectively. The reason for the almost 1:1 correlation is that the measured latency also includes the bootstrapping time.

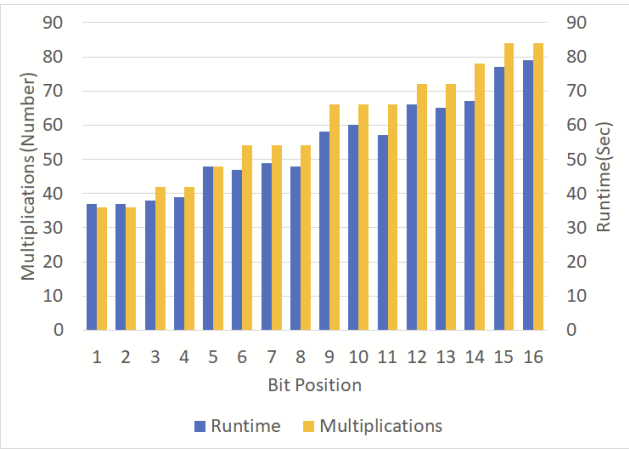


Figure 9: Cleanup time and number of non-scalar multiplications per bit when using $\text{CleanInteger}_{N=16,\alpha=0.1,\beta=10^{-3}}(\cdot)$.

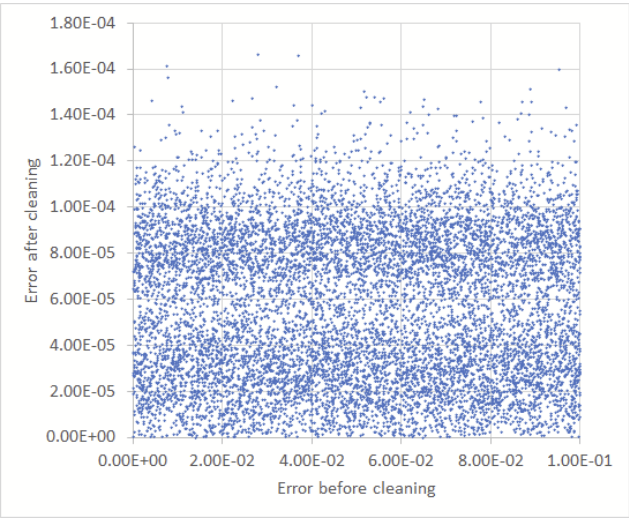


Figure 10: A correlation graph between the input error (X-axis) and the output errors after calling $\text{CleanInteger}_{N=16,\alpha=0.1,\beta=10^{-3}}(\cdot)$ (Y-axis). The 10,000 input errors were sampled from $[-0.1, +0.1]$.

As expected, there is an increment in the number of executed multiplications and measured latency for higher bits. The reason is that Algorithm 4 uses smaller precision parameters α, β for the more significant bits, resulting in a need for a more costly approximation for the sign function. In addition, the linear step-wise graph is the result of the sign approximation-polynomial degrees achieved from the lower bounds of [16][Corollary 3].

Using the data from Figure 9, we can estimate that cleaning 4-bit, 8-bit, and 16-bit integers will take around 160, 360, and 943 seconds, respectively

(resp. 2.5, 6, and 10 minutes). However, if we consider the amortized latency of performing 2^{14} cleanup using one ciphertext, we get 9.76, 21.9, and 57.55 msec, respectively, which makes our algorithm practical for some applications.

Finally, Figure 10 shows the correlation between the input error and the output errors in our experiments, where no special correlation was observed.

8 Applications

Obvious applications for BLEACH are Boolean computations and homomorphic branching using encrypted masks. We give two more applications that are less obvious: computing modulo and accessing a table.

8.1 Modulo

We show here how to implement the modulo operation on a noisy integer $m = x + e$, for $x \in \mathbb{N}$, $x < 2^{N+1}$ and $e \in \mathbb{R}$, i.e., compute $y = x \mod n$ for $n \in \mathbb{N}$. This leads to computation over $\mathbb{Z}_n$ natively in CKKS. This can be done by following these steps (we leave out the exact details):

- Set $v = \frac{x}{n} - \frac{n-1}{2n}$,
- $w = \left\lfloor \frac{x}{n} \right\rfloor$ is computed by $w := CleanInteger_{N, \frac{1}{2n}, \frac{1}{n}}(v)$,
- Compute the residue: $r = x - w \cdot n$.

To sketch a proof of the correctness (to simplify, assume $e = 0$), notice that

$$\left\lfloor \frac{x}{n} \right\rfloor \leq \frac{x}{n} \leq \left\lfloor \frac{x}{n} \right\rfloor + \frac{n-1}{n} \tag{22}$$

and so

$$\left\lfloor \frac{x}{n} \right\rfloor - \frac{1}{2} + \frac{1}{2n} = \left\lfloor \frac{x}{n} \right\rfloor - \frac{n-1}{2n} \leq \frac{x}{n} - \frac{n-1}{2n} = v \tag{23}$$

and also

$$v \leq \left\lfloor \frac{x}{n} \right\rfloor + \frac{n-1}{n} - \frac{n-1}{2n} = \left\lfloor \frac{x}{n} \right\rfloor + \frac{1}{2} - \frac{1}{2n}. \tag{24}$$

8.2 Table access

Table access is useful for example when a lookup table is precomputed to speed execution. In this case, we have a table of n entries $T[1], \ldots, T[n]$ and for an input index $x = 1, \ldots, n$ we want to output $T[x]$. When x is a ciphertext this can be done by:

$$y = \sum \mathrm{Cmp}(x, i) \cdot T[i],$$

where $\mathrm{Cmp}(a, b)$ is a function that returns 1 if $a = b$ and 0 otherwise. The accuracy of y, i.e. $|y - T[x]|$ depends on the accuracy of the function Cmp that needs to be more accurate (and therefore more expensive to compute) as

n grows. In this naive implementation, the size of the circuit that accesses a table is $O(nS_{\text{Cmp}})$, where S_{Cmp} is the size of the circuit implementing Cmp. We propose the following steps to access a table (again, we leave out the exact details):

- Compute the bit decomposition (b_j) of x.
- Set $y := \sum_i \text{CmpBin}((b_j), i) \cdot T[i]$, where CmpBin compares (b_j) to i in binary.

For large values of n, the implementation of CmpBin involves a circuit whose size is smaller than that of Cmp, i.e. $S_{Bin\,\text{Cmp}} < S_{\text{Cmp}}$. The size of the circuit that accesses a table is then:

$$O(S(Decompose) + n \cdot S_{Bin\,\text{Cmp}})),$$

where $S(Decompose)$ is the circuit size of Algorithm 3 and $S_{Bin\,\text{Cmp}} = O(\log n)$ is the size of the circuit that compares two n-bit numbers given in binary.

9 Discussion

In this paper, we chose specific demonstrators to show that an approximate scheme with SIMD capabilities such as CKKS can outperform exact schemes without SIMD capabilities such as TFHE. For example, Section 4 shows that every circuit that uses binary operations such as AND, OR, NOT, and XOR can be parallelized in a way that it acts similarly over a large enough number of inputs and that we can achieve an efficient implementation using CKKS. This left the interesting open question of identifing many such real-life circuits, and considering circuits that only involve some level of parallelization. In the latter case, we asked: can we reorganize these circuits by using some automatic compiler to increase their parallelization level?

One of the benefits of CKKS is that it operates over complex numbers. It is interesting to identify circuits that combine integer and complex values or even just integer and floating number operations. For example, XGBoost is a commonly used machine learning algorithm based on many decision trees with the same structure that runs in parallel. To train the model and also to infer data from it, its algorithm requires comparing floating-point values, generating binary indicator vectors, and summing them homomorphically. The leaves of the tree are again floating-point elements. The method we described in this paper should allow performing XGBoost training operation in CKKS, which was previously considered a hard task unless a client-aided design is used.

In many cases, companies and developers of cryptographic libraries prefer to maintain and support libraries with a small footprint. This reduces the number of potential bugs and also the entrance barriers for new developers. For that reason, we think it is important to allow developers to learn what they can achieve with an HE library that supports only one scheme, in our case, CKKS. We leave it as an open question to explore whether combining two schemes can achieve more efficient solutions. Perhaps this can be done,

for example, by transferring a CKKS ciphertext to TFHE, performing the cleanup (bootstrapping) in TFHE and returning the cleaned results to CKKS for further SIMD operations.

Deferring cleanups. In our experiments we BLEACHed ciphertexts after every operation. However, in some applications, it may be more efficient to defer BLEACHing until enough noise accumulates. Recall that the noise that builds up on an integer comes from two sources: the CKKS scheme noise and the noise coming from using polynomial approximations of non-polynomial functions. While the first type of noise is trackable, the latter requires knowledge of the circuit's functionality. We expect implementations of non-polynomial functions (e.g. ReLU, min, etc.) to also provide a guarantee of the accuracy of their output. This can be based on heuristic simulations or on theoretical analysis of the error by using e.g., the worst case analysis of [15] or the Central Limit Theorem (CLT)-based analysis of [20]. BLEACHing provides a systematic method to improve the accuracy guarantee of such approximations.

An Exact variant of CKKS. To protect against the recent key recovery attack against CKKS of Li and Micciancio [37], an exact variant of CKKS was presented in [20]. The [37] attack allows an adversary to use the decryption error to extract information about the secret key. Informally, the idea of [20] was to accurately track the error of a circuit, then define a circuit to be correct if its accumulated error does not interact with the message. For such correctable circuits, it is possible to apply the `Correct` method of [20] after the decoding operation. This operation cuts down the lowest bits of the plaintext before releasing it to the user. Thus, the adversary can't see the error and can't extract the private key. Our method can be used to generate a larger subset of correctable circuits for a smaller scale value, which can also result in future performance improvements.

10 Conclusion

We explored the feasibility of using an approximated HE scheme such as CKKS to perform deep or even unbounded circuits. We identified several scenarios such as wide binary circuits and the Game of Life that are easily parallelizable. For these cases, we showed that our implementations were not only feasible in terms of accuracy, but also efficient compared to other HE schemes that developers often automatically prefer for similar circuits.

This work revisits the question: which HE sub-primitive is the most important for achieving the efficiency of deep circuits? Is it an efficient bootstrapping capability? An efficient cleanup utility? Or the SIMD capability of the scheme? We believe that the process of finding the ultimate HE scheme is only in its infancy and we expect that the answers to these questions will change many times in the future. Still, today, our paper shows (maybe again) that for some use cases, good SIMD capabilities overcome the latency issues of a not-too-fast cleanup method.

References

[1] Adam, L.: BoringSSL (Oct 2015), https://www.imperialviolet.org/2015/10/17/boringssl.html

[2] Aharoni, E., Adir, A., Baruch, M., Drucker, N., Ezov, G., Farkash, A., Greenberg, L., Masalha, R., Moshkowich, G., Murik, D., Shaul, H., Soceanu, O.: HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data. CoRR **abs/2011.0** (2020), https://arxiv.org/abs/2011.01805

[3] Akavia, A., Vald, M.: On the privacy of protocols based on cpa-secure homomorphic encryption. IACR Cryptol. ePrint Arch. **2021**, 803 (2021), https://eprint.iacr.org/2021/803

[4] Arita, S., Nakasato, S.: Fully homomorphic encryption for point numbers. In: Chen, K., Lin, D., Yung, M. (eds.) Information Security and Cryptology. pp. 253–270. Springer International Publishing, Cham (2017). doi:https://doi.org/10.1007/978-3-319-54705-3_16

[5] Badawi, A.A., Bates, J., Bergamaschi, F., Cousins, D.B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., Liu, Z., Micciancio, D., Quah, I., Polyakov, Y., R.V., S., Rohloff, K., Saylor, J., Suponitsky, D., Triplett, M., Vaikuntanathan, V., Zucca, V.: OpenFHE: Open-Source Fully Homomorphic Encryption Library. Cryptology ePrint Archive, Paper 2022/915 (2022), https://eprint.iacr.org/2022/915

[6] Bae, Y., Cheon, J.H., Cho, W., Kim, J., Kim, T.: META-BTS: Bootstrapping Precision Beyond the Limit. Cryptology ePrint Archive, Paper 2022/1167 (2022), https://eprint.iacr.org/2022/1167

[7] Boemer, F., Cammarota, R., Demmler, D., Schneider, T., Yalame, H.: MP2ML: A Mixed-Protocol Machine Learning Framework for Private Inference. In: Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice. p. 43–45. PPMLP'20, Association for Computing Machinery, New York, NY, USA (2020). doi:https://doi.org/10.1145/3411501.3419425

[8] Boemer, F., Costache, A., Cammarota, R., Wierzynski, C.: NGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data. In: Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography. pp. 45–56. WAHC'19, Association for Computing Machinery, New York, NY, USA (2019). doi:https://doi.org/10.1145/3338469.3358944

[9] Bossuat, J.P., Mouchet, C., Troncoso-Pastoriza, J., Hubaux, J.P.: Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys. In: Canteaut, A., Standaert, F.X. (eds.) Advances in Cryptology – EUROCRYPT 2021. pp. 587–617. Springer International Publishing, Cham (2021). doi:https://doi.org/10.1007/978-3-030-77870-5_21

[10] Boura, C., Gama, N., Georgieva, M., Jetchev, D.: CHIMERA: Combining Ring-LWE-based Fully Homomorphic Encryption Schemes. Journal of Mathematical Cryptology **14**(1), 316–338 (2020). doi:https://doi.org/doi:10.1515/jmc-2019-0026

[11] Brakerski, Z.: Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology – CRYPTO 2012. vol. 7417 LNCS, pp. 868–886. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). doi:https://doi.org/10.1007/978-3-642-32009-5_50

[12] Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) Fully Homomorphic Encryption without Bootstrapping. ACM Transactions on Computation Theory **6**(3) (jul 2014). doi:https://doi.org/10.1145/2633600

[13] Chen, H., Chillotti, I., Song, Y.: Improved Bootstrapping for Approximate Homomorphic Encryption. In: Ishai, Y., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2019. pp. 34–54. Springer International Publishing, Cham (2019). doi:https://doi.org/10.1007/978-3-030-17656-3_2

[14] Chen, H., Han, K.: Homomorphic Lower Digits Removal and Improved FHE Bootstrapping. In: Nielsen, J.B., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2018. pp. 315–337. Springer International Publishing, Cham (2018). doi:https://doi.org/10.1007/978-3-319-78381-9_12

[15] Cheon, J., Kim, A., Kim, M., Song, Y.: Homomorphic Encryption for Arithmetic of Approximate Numbers. In: Proceedings of Advances in Cryptology - ASIACRYPT 2017. pp. 409–437. Springer Cham (11 2017). doi:https://doi.org/10.1007/978-3-319-70694-8_15

[16] Cheon, J.H., Kim, D., Kim, D.: Efficient homomorphic comparison methods with optimal complexity. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 221–256. Springer (2020). doi:https://doi.org/10.1007/978-3-030-64834-3_8

[17] Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology – ASIACRYPT 2016. pp. 3–33. Springer Berlin Heidelberg, Berlin, Heidelberg (2016). doi:https://doi.org/10.1007/978-3-662-53887-6_1

[18] Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: Fast Fully Homomorphic Encryption Over the Torus. Journal of Cryptology **33**(1), 34–91 (2020). doi:https://doi.org/10.1007/s00145-019-09319-x

[19] Chillotti, I., Joye, M., Ligier, D., Orfila, J.B., Tap, S.: CONCRETE: Concrete Operates oN Ciphertexts Rapidly by Extending TfhE. In: WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography. vol. 15 (2020)

[20] Costache, A., Curtis, B.R., Hales, E., Murphy, S., Ogilvie, T., Player, R.: On the precision loss in approximate homomorphic encryption. Cryptology ePrint Archive, Paper 2022/162 (2022), https://eprint.iacr.org/2022/162

[21] CryptoLab: HEaaN: Homomorphic Encryption for Arithmetic of Approximate Numbers, version 3.1.4 (2022), https://www.cryptolab.co.kr/eng/product/heaan.php

[22] Fan, J., Vercauteren, F.: Somewhat Practical Fully Homomorphic Encryption. Proceedings of the 15th international conference on Practice and Theory in Public Key Cryptography pp. 1–16 (2012), https://eprint.iacr.org/2012/144

[23] Florent, M.: Game of life using fully homomorphic encryption commit 04b7deebd9b96b2701c13e2d08c141b84f1c8479 (2022), https://github.com/FlorentCLMichel/homomorphic_game_of_life

[24] Gardner, M.: The fantastic combinations of John Conway's new solitaire game "life" (Oct 1970), https://www.scientificamerican.com/article/mathematical-games-1970-10/

[25] Gentry, C.: A Fully Homomorphic Encryption Scheme. Ph.D. thesis, Stanford University (2009), https://crypto.stanford.edu/craig

[26] Gentry, C., Halevi, S., Smart, N.P.: Better bootstrapping in fully homomorphic encryption. In: International Workshop on Public Key Cryptography. pp. 1–16. Springer (2012). doi:https://doi.org/10.1007/978-3-642-30057-8_1

[27] Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In: International Conference on Machine Learning. pp. 201–210 (2016), http://proceedings.mlr.press/v48/gilad-bachrach16.pdf

[28] Halevi, S., Shoup, V.: Bootstrapping for HElib. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology – EUROCRYPT 2015. pp. 641–670. Springer Berlin Heidelberg, Berlin, Heidelberg (2015). doi:https://doi.org/10.1007/978-3-662-46800-5_25

[29] Jiang, X., Kim, M., Lauter, K., Song, Y.: Secure outsourced matrix computation and application to neural networks. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. p. 1209–1222. CCS '18, New York, NY, USA (2018). doi:https://doi.org/10.1145/3243734.3243837, https://doi.org/10.1145/3243734.3243837

[30] Jutla, C.S., Manohar, N.: Sine Series Approximation of the Mod Function for Bootstrapping of Approximate HE. In: Dunkelman, O., Dziembowski, S. (eds.) Advances in Cryptology – EUROCRYPT 2022. pp. 491–520. Springer International Publishing, Cham (2022). doi:https://doi.org/10.1007/978-3-031-06944-4_17

[31] Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: GAZELLE: A low latency framework for secure neural network inference. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 1651–1669. USENIX Association, Baltimore, MD (Aug 2018), https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar

[32] Kim, A., Papadimitriou, A., Polyakov, Y.: Approximate homomorphic encryption with reduced approximation error. In: Cryptographers' Track at the RSA Conference. pp. 120–144. Springer (2022). doi:https://doi.org/10.1007/978-3-030-95312-6_6

[33] Lee, E., Lee, J.W., Kim, Y.S., No, J.S.: Minimax approximation of sign

function by composite polynomial for homomorphic comparison. IEEE Transactions on Dependable and Secure Computing (2021). doi:https://doi.org/10.1109/TDSC.2021.3105111

[34] Lee, J., Lee, E., Lee, J.W., Kim, Y., Kim, Y.S., No, J.S.: Precise approximation of convolutional neural networks for homomorphically encrypted data. arXiv preprint arXiv:2105.10879 (2021)

[35] Lee, Y., Lee, J.W., Kim, Y.S., Kim, Y., No, J.S., Kang, H.: High-Precision Bootstrapping for Approximate Homomorphic Encryption by Error Variance Minimization. In: Dunkelman, O., Dziembowski, S. (eds.) Advances in Cryptology – EUROCRYPT 2022. pp. 551–580. Springer International Publishing, Cham (2022). doi:https://doi.org/10.1007/978-3-031-06944-4_19

[36] Lehmkuhl, R., Mishra, P., Srinivasan, A., Popa, R.A.: Muse: Secure inference resilient to malicious clients. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 2201–2218. USENIX Association (Aug 2021), https://www.usenix.org/conference/usenixsecurity21/presentation/lehmkuhl

[37] Li, B., Micciancio, D.: On the security of homomorphic encryption on approximate numbers. In: Canteaut, A., Standaert, F.X. (eds.) Advances in Cryptology – EUROCRYPT 2021. pp. 648–677. Springer International Publishing, Cham (2021)

[38] Liu, J., Juuti, M., Lu, Y., Asokan, N.: Oblivious Neural Network Predictions via MiniONN Transformations. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 619–631. CCS '17, Association for Computing Machinery, New York, NY, USA (2017). doi:https://doi.org/10.1145/3133956.3134056

[39] Lou, Q., Jiang, L.: HEMET: A Homomorphic-Encryption-Friendly Privacy-Preserving Mobile Neural Network Architecture. In: Meila, M., Zhang, T. (eds.) Proceedings of the 38th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 139, pp. 7102–7110. PMLR (2021), https://proceedings.mlr.press/v139/lou21a.html

[40] Lu, W.j., Huang, Z., Hong, C., Ma, Y., Qu, H.: PEGASUS: Bridging Polynomial and Non-polynomial Evaluations in Homomorphic Encryption. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1057–1073 (2021). doi:https://doi.org/10.1109/SP40001.2021.00043

[41] Micciancio, D., Polyakov, Y.: Bootstrapping in FHEW-like Cryptosystems. In: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography. p. 17–28. WAHC '21, Association for Computing Machinery, New York, NY, USA (2021). doi:https://doi.org/10.1145/3474366.3486924

[42] Michel, F., Wilson, J., Cottle, E.: Concrete Boolean and Conway's Game of Life: A Tutorial (Oct 2021), https://medium.com/zama-ai/concrete-boolean-and-conways-game-of-life-a-tutorial-f2bcfd614131

[43] Michel, F., Wilson, J., Cottle, E.: Fully Homomorphic Encryption and the

Game of Life (Oct 2021), https://medium.com/optalysys/fully-homomorphic-encryption-and-the-game-of-life-d7c37d74bbaf

[44] Mishra, P., Lehmkuhl, R., Srinivasan, A., Zheng, W., Popa, R.A.: Delphi: A Cryptographic Inference Service for Neural Networks. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2505–2522. USENIX Association (aug 2020). doi:https://doi.org/10.1145/3411501.3419418, https://www.usenix.org/conference/usenixsecurity20/presentation/mishra

[45] Mohassel, P., Rindal, P.: ABY3: A Mixed Protocol Framework for Machine Learning. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 35–52. CCS '18, Association for Computing Machinery, New York, NY, USA (2018). doi:https://doi.org/10.1145/3243734.3243760

[46] Mohassel, P., Zhang, Y.: SecureML: A system for scalable privacy-preserving machine learning. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 19–38 (2017). doi:https://doi.org/10.1109/SP.2017.12

[47] Rathee, D., Rathee, M., Kumar, N., Chandran, N., Gupta, D., Rastogi, A., Sharma, R.: CrypTFlow2: Practical 2-Party Secure Inference. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 325–342. Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3372297.3417274

[48] Rendell, P.: Turing Universality of the Game of Life, pp. 513–539. Springer London, London (2002). doi:https://doi.org/10.1007/978-1-4471-0129-1_18

[49] Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (aug 2018). doi:https://doi.org/10.17487/RFC8446, https://rfc-editor.org/rfc/rfc8446.txt

[50] Rescorla, E., Dierks, T.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Aug 2008). doi:https://doi.org/10.17487/RFC5246, https://www.rfc-editor.org/info/rfc5246

[51] Riazi, M.S., Samragh, M., Chen, H., Laine, K., Lauter, K., Koushanfar, F.: XONN: XNOR-based oblivious deep neural network inference. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1501–1518. USENIX Association, Santa Clara, CA (Aug 2019), https://www.usenix.org/conference/usenixsecurity19/presentation/riazi

[52] Zama: fhe_game_of_life commit 6d15153ac234482f8b70841e5151a1a98cfc2775 (2022), https://github.com/zama-ai/fhe_game_of_life

Appendix A Optimal 7 bits cleanup utility.

Tables A1 - A7 show the optimal Minmax polynomial chains for cleaning up seven bits integers generated using [33][Alg. 6]. As explained therein even coefficients are set to 0. We experimented with this polynomial over plaintext values and observed that the cleanup algorithm went well. However, when we used them in CKKS the results were not quite good. One reason can be the floating point precision error of our implementation of the Remez algorithm.

However, because the results were good on plaintexts, we believe that the issue was related to the use of polynomials of large degrees, which by themselves add too much noise in CKKS.

Table A1: Coefficients of the polynomials in the polynomial chain of the 2nd bit cleanup function, where $\alpha = 0.1$ and $\beta = 0.01$ generated by our implementation of [33][Alg. 6]. Even coefficients are set to 0.

Polynomial	Coefficients
P_0	$a_1 = 6.255103778441478$ $a_3 = -59.396810504678996$ $a_5 = 456.5779496814207$ $a_7 = -2488.7300052851133$ $a_9 = 9821.21093546806$ $a_{11} = -28642.96196951083$ $a_{13} = 62487.65828476454$ $a_{15} = -102341.23929053893$ $a_{17} = 125187.33106336177$ $a_{19} = -112614.20722791724$ $a_{21} = 72278.73892266018$ $a_{23} = -31309.460586904195$ $a_{25} = 8199.391258797072$ $a_{27} = -980.1676282594433$

Table A2: Coefficients of the polynomials in the polynomial chain of the 2nd bit cleanup function, where $\alpha = 0.1$ and $\beta = 0.01$ generated by our implementation of [33][Alg. 6]. Even coefficients are set to 0.

Polynomial	Coefficients
P_0	$a_1 = 4.716693160325245$ $a_3 = -15.67353673707186$ $a_5 = 23.283939321409385$ $a_7 = -11.431278032966942$
P_1	$a_1 = 2.94967287170158$ $a_3 = -5.951799004595778$ $a_5 = 8.97420286218678$ $a_7 = -8.560130748648541$ $a_9 = 4.983074017699619$ $a_{11} = -1.6216075321501815$ $a_{13} = 0.2265875489536541$

Table A3: Coefficients of the polynomials in the polynomial chain of the 3rd bit cleanup function, where $\alpha = 0.1$ and $\beta = 0.01$ generated by our implementation of [33][Alg. 6]. Even coefficients are set to 0.

Polynomial	Coefficients
P_0	$a_1 = 9.695606891609275$ $a_3 = -144.4078405541289$ $a_5 = 1137.4333784916066$ $a_7 = -4606.759869925055$ $a_9 = 10205.156153439064$ $a_{11} = -12504.539801994544$ $a_{13} = 7949.963052602231$ $a_{15} = -2045.6360598517972$
P_1	$a_1 = 3.1572847817236087$ $a_3 = -7.423792920389324$ $a_5 = 13.430572003576687$ $a_7 = -16.027824094070652$ $a_9 = 12.463971519494718$ $a_{11} = -6.101774208186316$ $a_{13} = 1.7118342803420215$ $a_{15} = -0.21027136507338362$

Table A4: Coefficients of the polynomials in the polynomial chain of the 4th bit cleanup function, where $\alpha = 0.1$ and $\beta = 0.01$ generated by our implementation of [33][Alg. 6]. Even coefficients are set to 0.

Polynomial	Coefficients
P_0	$a_1 = 2.633048226501171$ $a_3 = -3.1014360529047176$ $a_5 = 1.778319169827171$ $a_7 = -0.350096986477881$
P_1	$a_1 = 2.9365692997589243$ $a_3 = -5.885242645281732$ $a_5 = 8.838577031033392$ $a_7 = -8.420786543062263$ $a_9 = 4.909784424651713$ $a_{11} = -1.6047198924868964$ $a_{13} = 0.22581832543195998$

Table A5: Coefficients of the polynomials in the polynomial chain of the 5th bit cleanup function, where $\alpha = 0.1$ and $\beta = 0.01$ generated by our implementation of [33][Alg. 6]. Even coefficients are set to 0.

Polynomial	**Coefficients**
P_0	$a_1 = 9.83592981440395$ $a_3 = -54.29469345630977$ $a_5 = 98.52897988979394$ $a_7 = -53.794786899779936$
P_1	$a_1 = 4.678757059465715$ $a_3 = -18.780014749604693$ $a_5 = 44.15718879154067$ $a_7 = -55.9791650652788$ $a_9 = 39.77887683259873$ $a_{11} = -15.855793274099822$ $a_{13} = 3.307774099271241$ $a_{15} = -0.2808818392031$
P_2	$a_1 = 3.144762123857163$ $a_3 = -7.347725986914387$ $a_5 = 13.237738648715732$ $a_7 = -15.76605046418342$ $a_9 = 12.262160795477353$ $a_{11} = -6.016680872610989$ $a_{13} = 1.695409026444237$ $a_{15} = -0.20961327078830438$
P_3	$a_1 = 8.404780976170263$ $a_3 = -43.14799325089339$ $a_5 = 76.49852865842198$ $a_7 = -41.290183864127926$

Table A6: Coefficients of the polynomials in the polynomial chain of the 6th bit cleanup function, where $\alpha = 0.1$ and $\beta = 0.01$ generated by our implementation of [33][Alg. 6]. Even coefficients are set to 0.

Polynomial	Coefficients
P_0	$a_1 = 10.755621479531257$ $a_3 = -61.50855864488836$ $a_5 = 112.8314847245435$ $a_7 = -61.92746762005894$
P_1	$a_1 = 6.166324355600753$ $a_3 = -31.554950092911813$ $a_5 = 78.34482892781759$ $a_7 = -96.79366851649982$ $a_9 = 64.41113044381709$ $a_{11} = -23.511867646324983$ $a_{13} = 4.431021349913756$ $a_{15} = -0.3368980534346619$
P_2	$a_1 = 3.1918454690036078$ $a_3 = -7.635691449054804$ $a_5 = 13.970257365432655$ $a_7 = -16.76054691100659$ $a_9 = 13.026493977239493$ $a_{11} = -6.337070804959791$ $a_{13} = 1.7567925053213245$ $a_{15} = -0.21208040484242466$
P_3	$a_1 = 2.9484438272009497$ $a_3 = -5.945542238467626$ $a_5 = 8.961443110742065$ $a_7 = -8.547029593460577$ $a_9 = 4.976196674232597$ $a_{11} = -1.6200272524938202$ $a_{13} = 0.22651548351103323$

Table A7: Coefficients of the polynomials in the polynomial chain of the 7th bit cleanup function, where $\alpha = 0.1$ and $\beta = 0.01$ generated by our implementation of [33][Alg. 6]. Even coefficients are set to 0.

Polynomial	Coefficients
P_0	$a_1 = 11.281995716622621$ $a_3 = -65.6498506039935$ $a_5 = 121.05323923447669$ $a_7 = -66.6060806333924$
P_1	$a_1 = 4.741946470078303$ $a_3 = -6.860289745581253$ $a_5 = 3.339005511684397$ $a_7 = -0.4916032445491064$
P_2	$a_1 = 2.8723824718256283$ $a_3 = -3.569637451924362$ $a_5 = 1.992939277196903$ $a_7 = -0.36845270483573317$
P_3	$a_1 = 2.9484438272009497$ $a_3 = -5.945542238467626$ $a_5 = 8.961443110742065$ $a_7 = -8.547029593460577$ $a_9 = 4.976196674232597$ $a_{11} = -1.6200272524938202$ $a_{13} = 0.22651548351103323$