Head Start: Digit Extraction in TFHE from MSB to LSB

Jan-Pieter D'Anvers*[1,2], Xander Pottier*[1], Thomas de Ruijter*[1], Ingrid Verbauwhede[1,2]

[1] COSIC KU Leuven, Leuven, Belgium, {firstname.lastname}@esat.kuleuven.be

[2] Belfort, Leuven, Belgium, {firstname.lastname}@belfortlabs.com

Abstract. TFHE bootstrapping is typically limited to a small plaintext space, with an exponential increase in cost for larger plaintext spaces. To bootstrap larger integers, one can use digit decomposition, a procedure that iteratively extracts and bootstraps a part of the larger plaintext space. Conventional state-of-the-art methods typically extract bits starting from the least significant bits (LSBs) and progress to the most significant bits (MSBs). However, we introduce a *DirtyMSB extraction* procedure that enables the digit decomposition from MSBs to LSB for the first time. However, this procedure introduces a small error during the extraction procedure. We demonstrate how to compensate this error in subsequent iterations. Compared to traditional LSB-to-MSB digit decomposition, our method improves the throughput, with for example an increase of 20% for a 5-bit plaintext and 50% increase for an 8-bit plaintext. In contrast to LSB-to-MSB methods, our extracted output ciphertexts have fresh noise, allowing us to directly use the extracted outputs for further computation without the need for an additional bootstrap or less efficient parameters. We demonstrate the applicability of our method by improving large-scale addition and scalar multiplication. Our method is particularly effective for vector addition operations, accelerating the addition of 1000 16-bit numbers by a factor of $\times 2.75$. Furthermore, we demonstrate a $\times 2.27$ speedup over the state-of-the-art implementation of scalar multiplication.

Keywords: FHE · TFHE · Bootstrapping · Vector Addition

1 Introduction

By enabling computations directly on encrypted data, Fully Homomorphic Encryption (FHE) allows sensitive information to be processed without exposing it. Therefore, this technology unlocks the ability to securely analyse private data in the cloud. For instance, it enables the computation on financial data to detect fraud [Max21], and on medical records to conduct medical research [ZLS+19]. Additionally, FHE can facilitate secure collaboration among multiple data holders, such as hospitals or research institutions, allowing them to jointly train machine learning models or conduct analyses without revealing their underlying, sensitive datasets [GGP+23].

Several FHE schemes have been developed to support computations on encrypted data. These schemes are generally categorised into two broad classes. The first class includes schemes that support batched computations, allowing for SIMD-style (Single Instruction, Multiple Data) parallelism. However, these schemes typically incur substantial computational and memory overhead, particularly during bootstrapping, a critical procedure used to

*Jan-Pieter D'Anvers, Xander Pottier and Thomas de Ruijter contributed equally to this work.

manage ciphertext noise growth. Notable schemes in this category include BGV [BGV14], BFV [FV12], CKKS [CKKS17] and GBFV [GV25].

The second class comprises schemes that operate on ciphertexts encrypting only a single, comparatively small integer each. This way, they offer significantly more lightweight ciphertexts and more efficient bootstraps at the cost of reduced parallelism. This class includes TFHE [CGGI16], FHEW [DM15] and FINAL [BIP+22]. Furthermore, during bootstrapping, these schemes can apply any function to the encrypted message by using programmable bootstrapping (PBS) as described by Chillotti et al. [CGGI16] and follow-up works [CGGI20, CJP21]. In this paper we will focus on this second class of FHE schemes and more specifically on the Torus Fully Homomorphic Encryption (TFHE) scheme [CGGI16].

Originally, TFHE bootstrapping worked on Boolean functions [DM15, CGGI20, BMMP18, ISZ19], which was later extended to support larger inputs/functions [CIM19]. Bergerat et al. [BBB+23] showed that the programmable bootstrapping procedure increases exponentially in cost with the number of plaintext input bits, with a maximal input plaintext of 11 bits. Lee and Yoon later introduced [LY23] a technique to increase the plaintext space, at the cost of extra blind rotation operations. Other techniques to increase the programmable bootstrapping size for general function evaluations include circuit bootstrapping [CLOT21, BBB+23, WWL+24, WHS+25, WBS+25] or tree-based bootstrapping [GBA21, TBC+25].

While general function evaluation is useful, it is typically costly and therefore overkill in some applications. One specific variant is digit decomposition, where a ciphertext with a large plaintext is bootstrapped and converted into multiple ciphertexts with smaller plaintext that together encode the original message. Chilotti et al. [CLOT21] and Liu et al. [LMP22] described a method to perform digit decomposition with a large plaintext space into multiple ciphertexts each encrypting a small chunk of the original message. Similar to the digit decomposition methods for BGV/BFV in [HS15, CH18], they start from the least significant bits (LSBs) and work their way to the most significant bits (MSBs). Crucially, their technique relies on a specific type of bootstrap that functions even with the padding bit filled. This method, known as WoP-PBS or Full Domain Functional Bootstrapping (FDFB), was introduced concurrently by Chillotti et al. [CLOT21] and Kluczniak and Schild [KS23], with subsequent improvements detailed in [YXS+21, LMP22, CZB+22, KS24].

Another approach to encode large messages is to use multiple ciphertexts with each a smaller plaintext size. Two main approaches have been proposed: a CRT-based approach by Kluczniak et Schild [KS23], and a radix-based approach [DM15, CGGI20] where a large integer m is split into blocks of less than β so that $m = \sum_i m_i \beta^i$. Bergerat et al. [BBB+23] introduced a methodology that splits a plaintext in a dedicated message and a carry space, where after bootstrapping the carry space is empty, and it gets filled in follow-up operations. This method is used for example in the implementation of the TFHE-rs library [Zam22].

Our contribution In this work, we improve digit decomposition, i.e., the extraction of the message encoded in a large plaintext space into multiple ciphertexts each encrypting a chunk of the original message. In contrast to previous methods, which perform this extraction from the least significant bits (LSBs) to the most significant bits (MSBs), we work the other way around, starting with the MSBs. This approach allows us to avoid a significant downside of prior work —the necessity of compensating for a filled padding bit— at the cost of making a small error that needs to be compensated later.

Specifically, we introduce the DirtyMSB extraction technique, which extracts the most significant bits of the message in a ciphertext up to a bounded error δ. This extraction technique only uses one programmable bootstrap and we perform an in-depth analysis of this bounded error. Using the DirtyMSB extraction as a building block, we construct a complete

and accurate digit decomposition algorithm that remains robust despite the imperfect nature of the DirtyMSB extraction.

Our method is cheaper than state-of-the-art methods for plaintext bit sizes larger than 3 (excluding padding bit), as we use only one bootstrapping operation to extract $\log_2(p)-2$ bits, whereas the state-of-the-art LSB-to-MSB extraction methods require two bootstraps to extract $\log_2(p)$ bits. Moreover, our method has the additional benefit that the outputs have lower noise, enabling us to directly use the outputs in further computations. Additionally, one can perform an approximate decomposition where only the most significant bits are extracted, which might be interesting for neural network applications where approximations are typically tolerated.

We also apply our method in the context of adding large batches of ciphertexts more efficiently, under the message-carry framework of [BBB+23]. The key idea is to convert ciphertexts from the standard message-carry representation to an extended plaintext representation with a larger plaintext space. Although this intermediate form lacks (efficient) programmable bootstrapping, it enables the summation of more ciphertexts before the plaintext space is full. At that point, the ciphertext needs to be reconverted into multiple ciphertexts with the original plaintext space to facilitate further operations.

We compare our method for various vector addition sizes and achieve a speedup of up to $\times 2.75$ compared to the efficient implementation in the TFHE-rs library [Zam22]. Furthermore, we demonstrate that this technique can be applied to other operations by combining our approach with a recent work by Pottier et al. [PDdRV25], which accelerates the (multi-)scalar multiplication. This further enhances the speed of the scalar multiplication operation: up to $\times 16.74$ compared to the TFHE-rs library and $\times 2.23$ compared to the work by Pottier et al. An implementation of our method is available on our Github repository [1].

2 Preliminaries

2.1 Notation

Let $\mathbb{Z}_q$ be the ring of integers modulo an integer q. In this paper, the modulus q is a power of two for simplicity, although the results of the paper can be generalised to other moduli. Let $x \xleftarrow{\$} \chi$ denote sampling an element x from a distribution χ, and let $x \xleftarrow{\$} \mathcal{U}(\mathbb{Z}_q)$ denote sampling the element x uniformly from the elements of $\mathbb{Z}_q$. Denote with $\lfloor x \rfloor$ ($\lceil x \rceil$) flooring (ceiling) the number x to the closest lower (higher) integer, and with $\lfloor x \rceil$ rounding the number x to the closest integer, rounding down in case of a tie.

2.2 Learning With Errors

A Learning With Errors (LWE) ciphertext encrypts a message $m \in \mathbb{Z}_p$ under a secret key $s \in \mathbb{Z}_q^n$, where $p|q$, by generating a uniformly random vector $a \xleftarrow{\$} \mathcal{U}(\mathbb{Z}_q^n)$ and a small error $e \xleftarrow{\$} \chi_e(\mathbb{Z}_q)$ from a small distribution χ_e. The ciphertext is then calculated as $ct = (a, b = a \cdot s + e + \Delta \cdot m)$, with $\Delta = q/p$. Decryption involves computing $b - a \cdot s$, which results in $e + \Delta \cdot m$, and rounding to the nearest multiple of Δ: $\lfloor (e + \Delta \cdot m)/\Delta \rceil$. Decryption is correct as long as the error e lies within the range $]-\Delta/2, \Delta/2]$.

LWE ciphertexts are additively homomorphic, meaning that two ciphertexts $ct_1 = (a_1, b_1 = a_1 \cdot s + e_1 + \Delta \cdot m_1)$ and $ct_2 = (a_2, b_2 = a_2 \cdot s + e_2 + \Delta \cdot m_2)$ can be added coefficient-wise to obtain:

$$ct_1 + ct_2 = (a_1 + a_2, b_1 + b_2 = (a_1 + a_2) \cdot s + e_1 + e_2 + \Delta \cdot (m_1 + m_2)), \tag{1}$$

[1] https://github.com/KULeuven-COSIC/Head_Start

which is a valid LWE ciphertext with larger noise e_1+e_2.

Multiple additions can be performed, and decryption remains correct under two conditions:

1. The messages do not overflow the plaintext space, meaning that the messages should be smaller than the plaintext size $p=q/\Delta$ to avoid overflows.
2. The total error e_{tot} remains small enough, specifically within the range $]-\Delta/2,\Delta/2]$.

2.3 Programmable Bootstrapping

As shown above, certain operations on ciphertexts increase the noise, which eventually can lead to incorrect decryption after a certain number of operations. To reset the noise to allow more operations to be performed, one can use a bootstrapping procedure. In this paper, we focus on third-generation schemes such as FHEW, TFHE, or FINAL, using TFHE bootstrapping as an example. We refer to [CLOT21] for details on the exact bootstrapping procedure. However, we highlight some details that are important for this paper.

In TFHE bootstrapping, a ciphertext $CT=(A,B=A\cdot s+E+\Delta\cdot m)$ with large noise E is transformed into a ciphertext $ct=(a,b=a\cdot s+e+\Delta\cdot m)$ encrypting the same message m with small noise e. Additionally, it is possible to compute any function on the input message essentially for free through a procedure called programmable bootstrapping (PBS). During a programmable bootstrap, one can choose any lookup table (LUT) that maps an input $i\in[0,p/2[$ to an output $\text{LUT}(i)$ and apply this LUT to the encrypted message while reducing the noise.

Programmable bootstraps are limited by two constraints. First, the most significant bit of the message, called the padding bit, should generally remain zero. Due to the negacyclic nature of the ring $R_q=\mathbb{Z}_q[X]/(X^N+1)$ (which means $a\cdot X^N=-a$), the lookup table has a negacyclic nature as well: $\text{LUT}(i)=-\text{LUT}(i-p/2)$ for $i\in[p/2,p[$. By keeping the padding bit zero, we only find values that were freely chosen and therefore implement the desired functionality. In special cases where the padding bit is one, the result for input values i larger than $p/2$ is $-\text{LUT}(i-p/2)$. By using WoP-PBS by Chillotti et al. [CLOT21] or FDFB by Kluczniak and Schild [KS23], one can overcome this first constraint to choose all outputs of the LUT, at the cost of additional operations.

Second, the plaintext space is typically limited. It has a maximum size of 11 bits, and is ideally chosen between 1 and 5 bits [BBB+23]. This means that we are limited to working with small integers in a single ciphertext. Programmable bootstraps are generally the most expensive operations in TFHE calculations. In this paper, we will use the number of PBS required as a benchmark for its efficiency.

Noise: After bootstrapping, the ciphertext contains an initial noise component denoted e_{br}, which originates from the blind rotations performed during the bootstrapping process. When multiple ciphertexts are added together, their respective noise terms accumulate, leading to an increase of the overall noise. The total noise present after these linear operations, prior to the next bootstrap, is denoted e_{lin}. The accumulated noise variance is expressed as:

$$\sigma^2_{\text{lin}}=C\cdot\sigma^2_{\text{br}},$$

where the coefficient C depends on the number and type of operations performed on the ciphertexts. For example, adding an independent, freshly bootstrapped ciphertext increases C by 1.

To maintain correctness, the value of C should be constrained by limiting the number of operations before triggering another bootstrap. In this paper, we follow the assumption of independence between bootstrapped ciphertexts ([CGGI16]; Assumption 3.11).

When bootstrapping a ciphertext, preprocessing operations such as key switching and modulus switching are performed before the noise is actually removed. These preprocessing

operations introduce their own noise, e_{ks} and e_{ms} respectively. Fortunately, these noise sources are not increased by adding ciphertexts together and only occur once in the noise term.

Bootstrapping is successful if the total noise $e_{\text{lin}} + e_{\text{ks}} + e_{\text{ms}}$ remains within the range $]-\Delta/2,\Delta/2]$. Typically, parameters are chosen so that this condition is met with overwhelming probability, for example an error probability of 2^{-64} or 2^{-128}.

ManyLUT: In some circumstances, it is possible to perform two lookups at the cost of only one PBS, as discussed in [CLOT21]. More specifically, we can perform a double lookup if one of the plaintext bits (excluding the padding) is guaranteed to be zero. This is typically the MSB or the LSB of the plaintext. In this case, we can perform a regular lookup, but knowing that this bit is zero, we use only half of the LUT. The remaining half of the LUT can be filled with a second function evaluation. After PBS, the value of the first LUT can be extracted as usual, and the value of the second LUT can be extracted by rotating the ciphertext by $2^i \cdot N/p$ positions, where i is the position of the zero bit in the plaintext. This technique allows for a double lookup, with the constraint that only half of the LUT is used for each function. We will refer to this technique as manyLUT.

2.4 Radix-based integer representations

Radix-based representations [DM15, CGGI20] divide larger messages into smaller chunks that are each encrypted in separate ciphertexts, called shortints. For example, a large message m is encrypted using radix β as:

$$\begin{gathered} \text{Enc}(m) = [ct_{\mathcal{B}},\ldots,ct_2,ct_1] \\ \text{where} \quad ct_i = \text{Enc}(m_i), \quad m_i \in [0,\beta-1], \quad \text{and} \quad m = \sum_{i=1}^{\mathcal{B}} m_i \beta^{i-1}. \end{gathered} \tag{2}$$

Linear computations can be performed on these encrypted large integers by considering each shortint ciphertext separately. For example, adding two integer ciphertexts can be achieved by adding the corresponding shortint ciphertexts.

Bergerat et al. [BBB+23] proposed a specific structure where some least significant bits of the plaintext space are designated message bits and are filled after a bootstrap, and the more significant bits, called the designated carry bits, are reserved to be filled during the linear operations. In this framework, after a certain number of additions, the carry space of the shortint ciphertexts becomes completely filled. A message-carry extraction is then performed, where each shortint ciphertext is split into two ciphertexts using PBS operations: one ciphertext encrypts the message, and the other encrypts the carry. This carry can then be added to the next more significant message block.

Vector Addition: An important operation on integer ciphertexts is vector addition, which is a foundation for tasks such as scalar multiplication and large-scale additions. These tasks are critical in applications such as neural network computations. Vector addition operates on $\mathcal{N}$ integers, each composed of $\mathcal{B}$ shortint ciphertexts. Initially, only the message spaces of these shortints are filled, with the carry spaces initialised to zero. The objective is to compute the sum of the $\mathcal{N}$ integers as efficiently as possible.

The process begins by grouping all shortint ciphertexts with the same significance in the encrypted large integers into blocks. Each block is then summed independently by adding shortints together until the plaintext is saturated. At that point, two programmable bootstrapping (PBS) operations are used to split the result into a message and a carry ciphertext. The extracted message ciphertext is added back into the current block, while the extracted

carry ciphertext is propagated to the next more significant block. When all blocks have been processed and all carries propagated, the final result is obtained.

Due to the small plaintext modulus, many message-carry operations are required. For instance, with a 4-bit plaintext size (excluding the padding bit), one can add 5 ciphertexts together to produce 2 new ciphertexts (the message and carry). This means that, at the cost of 2 PBS operations, the number of ciphertexts to be added is reduced by 3, resulting in an efficiency of 3/2. In this paper, we focus on increasing the plaintext space to reduce the computational cost of vector addition and similar operations.

3 General overview of our method

In this section we will focus on extracting (and bootstrapping) a ciphertext with a large integer plaintext into multiple ciphertexts each encrypting a chunk of the original message. Notably, we will work under the constraint that the extended plaintext space p_{ext} is larger than the original plaintext space p, that can be reliably bootstrapped under the chosen parameters.

Bootstrapping these extended-plaintext ciphertexts is not trivial, as we can only reliably bootstrap a $\log_2(p)$-bit plaintext (including the padding). Furthermore, the noise on this ciphertext must be small enough, as key switching and modulus switching introduce additional noise, which could lead to wrong bootstrapping results when combined with the present noise.

The state-of-the-art method approaches such an extraction from LSB to MSB, while we will show a method that works in the opposite direction.

3.1 LSB-first approach

In the LSB-first approach, the entire ciphertext is shifted upward by multiplying it with a factor of p_{ext}/p. This alignment makes sure that the least significant bits (LSBs) of the extended plaintext space correspond to the bootstrappable region of LWE ciphertexts. However, this shift can cause the padding bit to become non-zero, preventing the correct extraction of information using an identity LUT. To address this issue one can use a WoP-PBS/FDFB technique [CLOT21, KS23, YXS+21, LMP22, CZB+22, KS24] that allows bootstrapping with a filled padding bit.

The downside of these approaches is that they all use multiple PBS operations, and result in an increased output error. More specifically, WoP-PBS can correctly extract $\log_2(p)-1$ bits in two bootstrap operations (Algorithm 4 in [CLOT21]) or $\log_2(p)$ bits in three bootstrap operations (Algorithm 5 in [CLOT21]). Moreover, the extracted ciphertext in both cases will contain more noise compared to a ciphertext resulting from standard bootstrapping, as it is the result of an LWE multiplication. Similarly, FDFB uses two sequential bootstraps to extract $\log_2(p)$ bits (Algorithm 3 in [KS24]) and results in higher output noise as this ciphertext was rotated during two full blind rotation steps and multiplied by a polynomial encoding the LUT. The extraction procedure based on the homomorphic flooring by Liu et al. [LMP22] (Algorithm 4) uses two sequential bootstraps to extract $\log_2(p)$ bits. However, this procedure will pass on any noise from the input ciphertext to the extracted output ciphertexts. In conclusion, the LSB-first approach can extract $\log_2(p)$ bits in two bootstrap operations at best, for an average $2/\log_2(p)$ bootstraps per extracted bit. In this work, we aim to improve the efficiency and reset noise in the output ciphertexts.

3.2 MSB-first approach

Our MSB-first approach, on the other hand, attempts to bootstrap the $\log_2(p)-1$ most significant bits first (that is, the LWE plaintext space without padding bit). However, due to the pres-

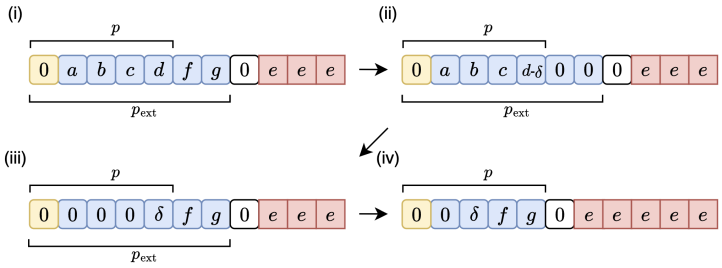

Figure 1: DirtyMSB approach: This multi-step process involves: (i) bootstrapping the $\log_2(p)-1$ most significant bits $(a\|b\|c\|d)$, resulting in (ii) a dirty ciphertext ct_{abcd^*} that may be off by a value δ. By subtracting this from the original ciphertext, we obtain (iii) a representation of the LSB plus δ. Finally, shifting this value up yields (iv) a bootstrappable ciphertext that can be processed with a regular PBS.

ence of unknown lower significant bits, and the additional noise introduced during key switching and modulus switching, this method fails to bootstrap these most significant bits correctly.

Our approach follows what we call a **DirtyMSB approach**. The main idea is to extract the $\log_2(p)-1$ most significant bits first in an approximate way, and only later account for any error δ that might have occurred. The procedure is illustrated intuitively in Figure 1, where we start from a ciphertext with an extended plaintext size p_{ext} (i). When extracting the MSBs $(a\|b\|c\|d)$, the result may be slightly inaccurate due to fg and the noise, yielding ciphertext ct_{abcd^*} (ii) encrypting value $(0\|a\|b\|c\|d)-\delta$. In the rest of the paper, we will denote with ct_{x^*} an approximate, 'dirty' version of ciphertext ct_x.

Next, we subtract the bootstrapped ciphertext (ii) from the original ciphertext (i), resulting in a new ciphertext (iii) that encrypts $(0\|0\|0\|0\|\delta\|f\|g)$. By shifting this ciphertext by a factor of p_{ext}/p, we align the remaining bits for a second PBS that extracts the LSBs: $(\delta\|f\|g)$.

The outcome of this procedure is two standard TFHE ciphertexts: one encrypting the MSBs minus δ, and the other encrypting the LSBs plus $\delta\cdot(p_{\text{ext}}/p)$. Together, these ciphertexts form a correct integer encryption of the original integer, now represented as two ciphertexts encrypting $m_{msb}-\delta$ and $m_{lsb}+\delta\cdot(p_{\text{ext}}/p)$:

$$(m_{\text{msb}}-\delta)\cdot\frac{p_{\text{ext}}}{p}+(m_{\text{lsb}}+\delta\cdot\frac{p_{\text{ext}}}{p})=m_{\text{msb}}\cdot\frac{p_{\text{ext}}}{p}+m_{\text{lsb}}=m.$$

The above procedure can also be applied to larger plaintext sizes p_{ext}, by applying the procedure iteratively. More specifically, after extracting a dirty version of the MSBs and subtracting this from the original ciphertext to find a new extended-plaintext ciphertext with MSBs reset to zero, we can shift it with $\log_2(p)-2$ bits, to obtain a new ciphertext on which we can apply the DirtyMSB technique again.

With our approach, it is possible to extract $\log_2(p)-1$ bits using only a single bootstrapping operation. However, since an error δ was introduced, only $\log_2(p)-2$ bits are removed from the original ciphertext. Consequently, we need an average of $1/(\log_2(p)-2)$ bootstraps per extracted bit. For instance, if $\log_2(p)=5$, a commonly used plaintext size in TFHE applications, this results in a theoretical 20% increased throughput compared to the LSB-first approach, while for $\log_2(p)=8$ we get a 50% increased throughput. Our technique also outputs 'clean' ciphertexts with low noise, resulting in more optimal parameter sets, as will be explained later. In the next section, we will give more details into the DirtyMSB technique and analyse the bounds of the error, δ.

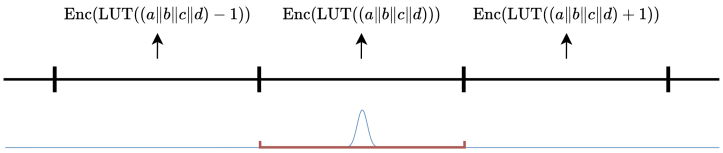

Figure 2: Visualisation of the error distribution during normal bootstrapping. The upper part indicates the resulting value from the bootstrap. The bottom part is the probability density distribution of the error, with the bounds that contain the error with overwhelming probability (e.g., $1-2^{-64}$) indicated in red.

4 Digit Decomposition using DirtyMSB Extraction

In this section, we will go into detail on the conversion of a ciphertext with an extended plaintext size p_{ext} to multiple ciphertexts with the original plaintext size p.

4.1 DirtyMSB bootstrap

First, we will examine what happens during the first DirtyMSB bootstrap. When bootstrapping a standard TFHE ciphertext, we operate on encryptions of $m\cdot\Delta+e$, where e is the noise. The scaling factor Δ provides robustness against this noise, and the parameters are chosen such that the error lies within the range $]-\Delta/2,\Delta/2]$ with high probability. This ensures that encryptions of $m\cdot\Delta+e$ are correctly mapped to encryptions of the lookup value $\text{LUT}(m)$, as shown in Figure 2.

However, in the extended plaintext case, the LSBs that do not fit into plaintext space p must also be considered. The bootstrapped value becomes:

$$m_{\text{msb}}\cdot\Delta+(m_{\text{lsb}}\cdot\Delta_{\text{ext}}+e),$$

where $\Delta=q/p$ and $\Delta_{\text{ext}}=q/p_{\text{ext}}$, and where m_{lsb} is unknown and not intended to influence the result. This new term m_{lsb} effectively acts as a second noise source. This noise source has a range:

$$[0,\max(m_{\text{lsb}}\cdot\Delta_{\text{ext}})[=[0,\Delta-\Delta_{\text{ext}}[=\left[0,\Delta\cdot\left(1-\frac{p}{p_{\text{ext}}}\right)\right[,$$

which tends to $[0,\Delta[$, in the worst-case scenario where p_{ext} is large. This means we have a noise sources with range $]-\Delta/2,\Delta/2]$ and one with $[0,\Delta[$, for a total noise range of $]-\Delta/2,\Delta\cdot3/2[$. This situation is illustrated in red in Figure 3. As a result, the output of the DirtyMSB bootstrap may deviate slightly, yielding $m_{\text{msb}^*}=m_{\text{msb}}-\delta$ with $\delta\in\{-1,0\}$.

This DirtyMSB value can then be subtracted from the original extended-plaintext ciphertext to isolate the remaining part of the message. The resulting ciphertext encrypts:

$$m\cdot\Delta_{\text{ext}}-m_{\text{msb}^*}\cdot\Delta=\left(m-(m_{\text{msb}}-\delta)\cdot\frac{p_{\text{ext}}}{p}\right)\cdot\Delta_{\text{ext}}=\left(m_{\text{lsb}}+\delta\cdot\frac{p_{\text{ext}}}{p}\right)\cdot\Delta_{\text{ext}}.$$

However, if $\delta=-1$, this subtraction causes an underflow. One trivial way to address this is to always add a constant scalar p_{ext}/p to the ciphertext before continuing, and compensate for this offset at the end of the conversion process.

To avoid dealing with this constant offset later, we can instead ensure that underflows during the DirtyMSB bootstrap are impossible. By subtracting Δ from the input extended-plaintext ciphertext before the DirtyMSB extraction, we can shift the noise range around the message from $]-\Delta/2,\Delta\cdot3/2[$ to $]-\Delta\cdot3/2,\Delta/2[$. This ensures that the error term δ can only take

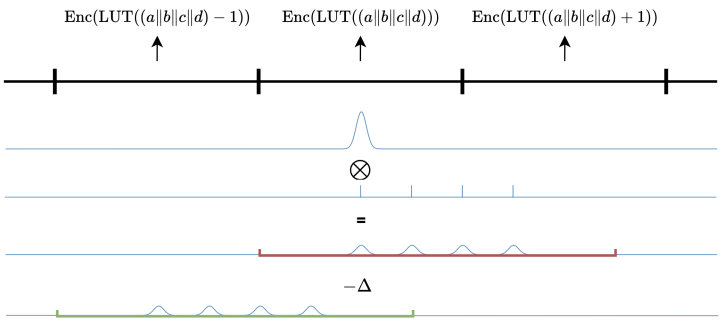

Figure 3: Visualisation of the error distribution during DirtyMSB bootstrapping. The upper part indicates the resulting value from the bootstrap. The bottom part is the probability density distribution of the error, with the bounds that contain the error with overwhelming probability (e.g., $1-2^{-64}$) indicated in red. The total noise now includes a contribution from the LSB and spans two LUT positions, potentially causing an error of $\delta=-1$. To prevent any underflows, Δ is subtracted such that the probability bounds are shifted, as indicated in green.

values 0 or 1, guaranteeing that the subtraction will not cause underflow. The bounds that contain the noise range after this subtraction are indicated in green in Figure 3.

After subtracting Δ from the original extended-plaintext ciphertext and applying the DirtyMSB extraction technique, we obtain a dirty value $m_{\text{msb}^*}=m_{\text{msb}}-\delta$, with $\delta\in\{0,1\}$, which is then subtracted from the original extended-plaintext ciphertext. This results in a new ciphertext encoding $m_{\text{lsb}}+\delta\cdot\frac{p_{\text{ext}}}{p}$. Together, these two ciphertexts represent the correct original message, since:

$$(m_{\text{msb}}-\delta)\cdot\frac{p_{\text{ext}}}{p}+(m_{\text{lsb}}+\delta\cdot\frac{p_{\text{ext}}}{p})=m.$$

This shows that the error introduced during the DirtyMSB extraction is fully compensated in the LSB component and the final result correctly reconstructs the original plaintext. On the other hand, as the error δ is now 0 or 1, an underflow can occur on the extracted MSB value $m_{msb^*}=m_{msb}-\delta$. In the next section, we will discuss how we can prevent this underflow using a special LUT setup.

4.2 Dealing with MSB underflows

In the previous section, we made sure that the subtraction of the dirty MSB from the original extended-plaintext ciphertext does not cause an underflow by ensuring that the error term $\delta\in\{0,1\}$. In this section, we examine possible underflows of the MSB value due to this error term.

When extracting the MSB, there is a realistic chance of finding an encryption of $m_{\text{msb}}-1$ due to the error δ in the extraction procedure. In general, this is not problematic, as the error is compensated by a corresponding increase in the LSBs. However, if the MSBs equal zero, this results in an underflow that should not occur. In this case, the underflow would turn on the padding bit and the extracted value would be incorrect when using the identity LUT. To address this, we will detect such underflows and explicitly bootstrap them to the correct value of 0, within the same bootstrapping procedure.

To correctly detect the underflow, our modified LUT must correctly map all regular values, including the underflowed value $0-\delta=-1$. In other words, besides the original $p/2$ values encoded in the LUT, we have to set one additional entry to a chosen value, resulting in a total of $p/2+1$ values.

Fortunately, we can exploit a concept from boolean TFHE bootstrapping, described by Chillotti et al. [CGGI16], to freely choose the result of bootstrapping any of $p/2+1$ input values by using a linear transformation. This concept is based on the observation that we can both choose the values of LUT[0] up to LUT$[p/2-1]$ and add a plaintext value to the result of that bootstrap. Moreover, we know that due to the negacyclic nature of the bootstrap, the bootstrapping result of the additional value that we want to choose wraps around as

$$\mathrm{LUT}[-1] = \mathrm{LUT}[p-1] = -\mathrm{LUT}[p/2-1].$$

To construct the desired LUT, we subtract a constant x from all entries of the LUT. After the bootstrap, we simply add this scalar x back to the result, which does not increase the noise as x is a plaintext value. When the desired result for input value $p/2-1$ is M, we can change the result of a lookup on input -1 to zero by choosing x such that:

$$\begin{cases} \mathrm{LUT}[p/2-1]+x & = M \\ \mathrm{LUT}[-1]+x = -\mathrm{LUT}[p/2-1]+x & = 0 \end{cases} \Rightarrow x = \frac{M}{2}.$$

Thus, by adapting the lookup values from LUT$[i]$ to LUT$[i] - \frac{M}{2}$ and adding $\frac{M}{2}$ to the bootstrapping result, we obtain an identity lookup that also correctly handles underflows.

4.3 General Digit Decomposition Procedure

To create a generalised conversion procedure, we define t_i as the number of MSBs that will be extracted from the current extended-plaintext ciphertext during iteration i of the procedure. It is important to note that, except for the first iteration, extracting t_i bits effectively retrieves only t_i-1 bits of the initial extended-plaintext ciphertext, since one of the extracted bits is replaced by the error term δ from the previous dirty extraction. Therefore, if k denotes the total number of iterations required to extract a complete set of ciphertexts from the extended-plaintext input, the values t_i must satisfy the following condition:

$$t_1 + \sum_{i=2}^{k} (t_i - 1) = \log_2\left(\frac{p_{\mathrm{ext}}}{2}\right). \tag{3}$$

Given that $t_i \leq \log_2(p/2)$, as we cannot bootstrap more bits than $\log_2(p/2)$ at once, the total number of iterations k is bounded by:

$$k \geq \left\lceil \frac{\log_2(p_{\mathrm{ext}}/2)-1}{\log_2(p/2)-1} \right\rceil$$

This iterative procedure for converting an extended-plaintext ciphertext into a list of regular-plaintext ciphertexts is summarised in algorithm 1. To illustrate how the algorithm operates in practice, Example 4.1 demonstrates the process using a 6-bit extended-plaintext ciphertext and $T = [2,3,3]$.

Details of the algorithm: In detail, line 2 first adds an offset to ensure $\delta = \{0,1\}$, as described in Subsection 4.1. Second, line 3 performs a lookup on the t_i most significant bits (MSBs). To understand this operation, we can interpret the value $(\log_2(p/2) - t_i)$ as the number of plaintext bits that are not converted at this stage.

Our lookup table evaluates $f(x) = \lfloor x \cdot 2^{-(\log_2(p/2)-t_i)} \rfloor$, which effectively performs an identity lookup ignoring the $(\log_2(p/2)-t_i)$ LSBs. To prevent underflows, we subtract $\frac{2^{t_i}-1}{2}$ from the LUT values and later re-add this plaintext value to the ciphertext, as explained in Subsection 4.2.

Algorithm 1: Conversion of one extended-plaintext ciphertext to a list of LWE ciphertexts

Input : $\hat{ct}$: An extended-plaintext ciphertext with plaintext size p_{ext}
Input : T: Vector of k integers t_i such that $t_1 + \sum_{i=2}^{k}(t_i - 1) = \log_2\left(\frac{p_{\text{ext}}}{2}\right)$
Output : List of regular-plaintext ciphertexts that represent the message of $\hat{ct}$

1 **for** $i = 1$ **to** $k-1$ **do**
2 $\quad \hat{ct}_{\text{offset}} \leftarrow \hat{ct} - \Delta$
3 $\quad ct_{i^*} \leftarrow PBS(\hat{ct}_{\text{offset}}, f(x) = \lfloor x \cdot 2^{-(\log_2(p/2) - t_i)} \rfloor - \frac{2^{t_i}-1}{2}) + \frac{2^{t_i}-1}{2}$ // DirtyMSB
4 $\quad ct_{i\_\text{shift}^*} \leftarrow 2^{\log_2(p/2) - t_i} \cdot ct_{i^*}$
5 $\quad$ **if** $i < k-1$ **then**
6 $\quad\quad \hat{ct} \leftarrow (\hat{ct} - ct_{i\_\text{shift}^*}) \cdot 2^{t_i - 1}$
7 $\quad$ **else**
8 $\quad\quad \hat{ct} \leftarrow (\hat{ct} - ct_{i\_\text{shift}^*}) \cdot 2^{t_k - (\log_2(p/2) - (t_i - 1))}$
9 $\quad$ **end**
10 **end**
11 $ct_k = PBS(\hat{ct}, f(x) = x)$
12 **return** $ct_{1^*}, \ldots, ct_{(k-1)^*}, ct_k$

Third, line 4 aligns the extracted MSBs with the MSBs of the initial ciphertext. Lines 5–9 then eliminate the MSBs and shift the extended-plaintext ciphertext upwards to reduce the plaintext size. This shift corresponds to the number of bits that have been effectively removed (i.e., $t_i - 1$), or in the final iteration, aligns the remaining LSB bits.

Example 4.1. Figure 4 illustrates the step-by-step process of the general conversion for a 6-bit extended-plaintext ciphertext. The superscripts on the arrows correspond to the lines in algorithm 1. The process begins by selecting values for t_i that satisfy the condition in Equation 3. Given that $\log_2(p_{\text{ext}}/2) = 6$, a valid choice is: $t_1 = 2$, $t_2 = 3$, $t_3 = 3$.

The conversion begins by subtracting Δ from the input ciphertext $\hat{ct}$. Next, our DirtyMSB operation extracts the $t_1 = 2$ MSBs, putting them in an LWE ciphertext as the least significant bits of the plaintext. The extracted ct_{1^*} is then shifted back up and subtracted from the input ciphertext $\hat{ct}$. Afterwards, the result is shifted up with a factor $2^{t_1 - 1} = 2$. The process then repeats for $t_2 = 3$. However, in the final iterating step, the result is multiplied by a different number ($2^{t_3 - (\log_2(p/2) - (t_2 - 1))} = 2^{3 - (4-2)} = 2$) since the remaining number of bits to extract ($t_3 = 3$) is smaller than $\log_2(p/2)$, thus a full shift to the MSB is unnecessary. In the last step, an identity bootstrap is performed, as there are no values that could introduce an additional error δ in the extracted result.

4.4 Noise analysis

In this section, we will look at the noise requirements of our method. First of all, one can clearly see that the noise in the output ciphertexts is the same noise as if they would be generated through a normal bootstrap, that is, the noise is effectively reset.

However, our method does have slightly stricter requirements on the input noise as the LSB-first techniques.

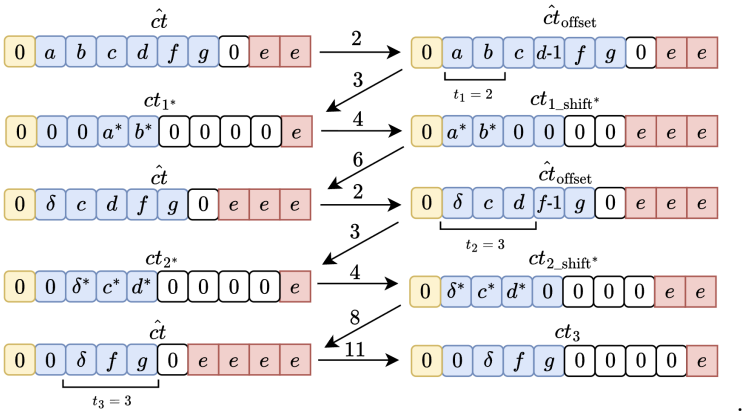

Figure 4: Example of the conversion of a 6-bit extended-plaintext ciphertext to a list of regular-plaintext ciphertexts. The numbers correspond to the lines of algorithm 1

DirtyMSB: For the DirtyMSB procedure, the noise analysis is relatively straightforward. The DirtyMSB operation tolerates a small error $\delta \in \{0,1\}$, which means that the noise must be within the interval $]-\Delta\cdot 3/2, \Delta/2]$. As discussed in Subsection 4.1, this condition is met when an LWE ciphertext with identical noise e and the same message m_{msb} but without the additional message bits outside of the plaintext space bootstraps correctly. This requirement is less stringent than that of the final LSB extraction and, therefore, does not constitute the most critical noise constraint.

Last (exact) bootstrap noise: To extract the LSBs in the final step, we perform a standard programmable bootstrap on a ciphertext encrypting $m\cdot\Delta+e$, where e represents the total noise accumulated during the previous operations and MSB extraction steps. However, in this context, the error e is larger than in typical LWE scenarios.

The variance of this error e comprises three main components. The first two are the pre-processing noises of the PBS, that is, the modulus switching noise variance σ^2_{ms} and the key switching noise variance σ^2_{ks}. The third component is the noise variance of the ciphertext $\hat{ct}$ that is input into the bootstrap at line 11. We will call this noise σ^2_{ext}. The total noise variance can thus be expressed as:

$$\sigma^2_e = \sigma^2_{\text{ms}} + \sigma^2_{\text{ks}} + \sigma^2_{\text{ext}}. \tag{4}$$

We now express σ^2_{ext} in terms of the noise after a bootstrap: $\sigma^2_{\text{ext}} = C_{\text{ext}}\cdot\sigma^2_{\text{br}}$. Larger values of C_{ext} require adjustments to the parameter set to ensure that the upper bound on the error probability remains satisfied. To determine suitable parameters, we employ the lattice estimator by Albrecht et al. [APS15], and incorporate the mean compensation technique proposed by de Ruijter et al. [dRDV25][2]. Throughout, we maintain a security of 128 bits and ensure a fault probability of at most 2^{-64}. As illustrated in Figure 5, the adjusted parameters do not significantly increase the bootstrapping time for a 4-bit plaintext space (excluding padding) as long as $C_{\text{ext}} \leq 1065$, after which the number of levels for blind rotation gadget decomposition is increased. These parameter sets are detailed in Appendix A.

During the digit decomposition, the noise in the main extended-plaintext ciphertext will grow. We will denote with $W\cdot\sigma^2_{\text{br}}$ the noise at the input of the digit decomposition. We will now work out the noise growth during digit decomposition by calculating the value of C_{ext}, based on an input noise multiplier W.

[2] https://github.com/KULeuven-COSIC/Head_Start

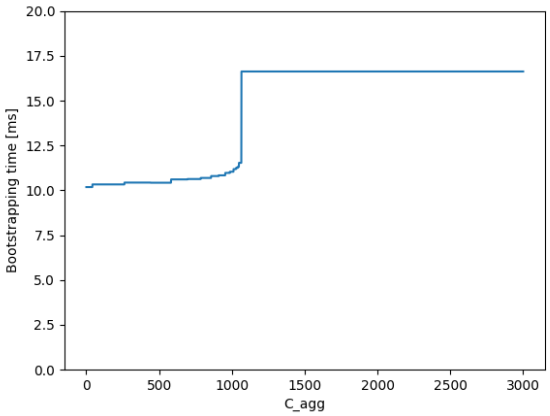

Figure 5: Latency of one bootstrap of a 4-bit plaintext space LWE ciphertext (excluding padding) using 1 thread measured in the TFHE-rs library with 128-bit security and error probability 2^{-64} on an AMD EPYC 9174F 16-core processor.

Digit decomposition noise growth: During the digit decomposition, we perform two types of operations that increase the noise variance. First, we subtract a shifted fresh ciphertext ($ct_{i\_\text{shift}^*}$), adding new noise to the extended-plaintext ciphertext. Since this ciphertext is the result of a bootstrap multiplied by a factor $2^{\log_2(p/2)-t_i}$, its noise variance equals $2^{2\cdot(\log_2(p/2)-t_i)}$. Second, we multiply the result of this subtraction by $2^{(t_i-1)}$, multiplying both the message and all present noise by a factor $2^{(t_i-1)}$, and thus amplifying the present noise variance with a factor $2^{2\cdot(t_i-1)}$. Consequently, the noise variance multiplier of the output of iteration i of the conversion procedure, C_{i+1}, is computed based on the noise variance multiplier of its input, C_i, as:

$$C_{i+1} \leftarrow (C_i + 2^{2\cdot(\log_2(p/2)-t_i)}) \cdot 2^{2\cdot(t_i-1)} \quad \text{with} \quad C_1 = W \quad \text{and} \quad 1 \leq i \leq k-2.$$

In iteration $k-1$ of the conversion procedure, we only multiply by $2^{t_k-(\log_2(p/2)-(t_{k-1}-1))}$, resulting in a noise variance change of:

$$C_k \leftarrow (C_{k-1} + 2^{2\cdot(\log_2(p/2)-t_{k-1})}) \cdot 2^{2\cdot(t_k-(\log_2(p/2)-(t_{k-1}-1)))}.$$

Combining these equations, and utilising Equation 3, we can derive the total noise variance multiplier C_{ext} for the final, critical extraction is:

$$C_{\text{ext}} = C_1 \cdot \left(2^{-2\log_2(p/2)} \cdot 2^{2t_k} \cdot \prod_{i=1}^{k-1} 2^{2\cdot(t_i-1)}\right) + \sum_{j=1}^{k-1} \left(2^{2\cdot(t_k+(t_{k-1}-1)-t_j)} \cdot \prod_{i=j}^{k-2} 2^{2\cdot(t_i-1)}\right) \tag{5}$$

$$= C_1 \cdot \left(\frac{p_{\text{ext}}}{p}\right)^2 + \sum_{j=1}^{k-1} 2^{\sum_{i=j+1}^{k} 2\cdot(t_i-1)}. \tag{6}$$

To gain a better understanding of this formula, we can express the number of plaintext bits (excluding the padding bit) in the TFHE ciphertext as $B = \log_2(p/2)$ and the additional bits in the extended plaintext as $E = \log_2(p_{\text{ext}}/p) = \log_2(p_{\text{ext}}/2) - B$. Moreover, we can see that the second term of the noise contains a sum off different powers of two, which by the geometric series formula is smaller than two times the largest power of two. Then, we can

bound the above formula by:

$$C_{\text{ext}} = C_1 \cdot 2^{2\cdot E} + \sum_{j=1}^{k-1} 2^{\sum_{i=j+1}^{k} 2\cdot(t_i-1)} \tag{7}$$

$$\leq W \cdot 2^{2\cdot E} + 2 \cdot 2^{2\cdot(E+B-t_1)} \tag{8}$$

$$\leq \left(W + 2^{2\cdot(B-t_1)+1}\right) \cdot 2^{2\cdot E}. \tag{9}$$

This term consists of two parts: a term consisting $W \cdot 2^{2\cdot E}$ which indicates the scaled input noise, and which is similar to the noise term that would be present in LSB to MSB conversion, and an additional noise term that is introduced during the digit decomposition of $2^{2\cdot(B-t_1)+1}$. In many cases, one would use the full plaintext to perform extraction, which results in $t_1 = B$ or an additional noise of 2 on top of the W noise:

$$C_{\text{ext}} \leq (W+2) \cdot 2^{2\cdot E}. \tag{10}$$

As such one can see that the overhead of our technique in terms of input noise is very limited.

4.5 Comparison of LSB-to-MSB versus MSB-to-LSB conversion

Table 1 compares the LSB-to-MSB extraction techniques from [CLOT21, LMP22, KS24] with our DirtyMSB extraction. First, we compare the upper bound of the input noise for the different techniques, with W the input noise multiplier. Using WoP-PBS or FDFB, no additional noise is added to the input ciphertext, resulting in the upper bound for C_{ext} of $W \cdot 2^{2E}$, as we are bootstrapping ciphertexts that encode E plaintext bits more than they can directly bootstrap. For the flooring based method from [LMP22], we find that one additional ciphertext is subtracted from the ciphertext we decompose to remove the padding bit. This results in an upper bound of $(W+1) \cdot 2^{2E}$. On the other hand, our suggested DirtyMSB extraction technique gives a slightly worse bound for the input noise with $(W+2) \cdot 2^{2E}$ as found in Subsection 4.4 compared to LSB-to-MSB approaches. However, the difference is limited since typically the input noise multiplier $W \gg 2$, as will be shown in Section 5.

Second, we compare the noise present on the output ciphertexts encrypting the different chunks of the extended plaintext. For WoP-PBS and FDFB, we take the formulas from [KS24], finding $O(N \cdot p) \cdot \sigma_{\text{br}}^2$ due to the LWE multiplication in WoP-PBS, and $O(p^3) \cdot \sigma_{\text{br}}^2$ due to the polynomial multiplication for FDFB. Notably, the flooring based method in [LMP22] does not remove the input noise from the output ciphertexts. The outputs are found by considering the inputs modulo a smaller q, equivalent to shifting up the ciphertext to remove the upper bits. As a result, this procedure leaves any input noise, by definition $W \cdot \sigma_{\text{br}}^2$, on the outputs. In contrast to all LSB-to-MSB extraction techniques, our outputs are the direct result of a bootstrap, with a significantly lower σ_{br}^2 output noise. This means that for all LSB-to-MSB conversions, if one wants to have 'bootstrap clean' noise in the outputs, one needs to perform one or more additional bootstrap operations.

Third, we consider the number of PBS operations for each of the methods. As this depends on the number of bits in the extended-plaintext encryption, we compare the number of PBS operations per extracted bit. We see that all LSB-to-MSB methods require multiple PBS per iteration, leading to a 2 term in the numerator, while our method only requires 1 PBS per iteration. However, we process two bits less than the state-of-the-art methods per iteration. In total, the cost of the state-of-the-art is $2/\log_2(p)$ while our method improves this to $1/(\log_2(p)-2)$.

Finally, the DirtyMSB extraction method allows to perform a cheaper 'approximate' decomposition on an extended-plaintext. For this, one performs an MSB extraction on only some of the most significant chunks, dropping some of the least significant bits. While this is not an exact computation, such approximate computations might be useful in some applications, such

Table 1: Comparison of the different digit decomposition approaches. We compare the input and output noise bounds, the cost, and the possibility to compute an approximate decomposition.

	Start. Point	Input noise	Output noise	Cost / extr. bit #PBS/bit	Appr. Decom.
[CLOT21]	LSB	$C_{\text{ext}} \leq W \cdot 2^{2E}$	$\mathcal{O}(N \cdot p) \cdot \sigma_{\text{br}}^2$	$\frac{2}{\log_2(p)-1}$	✗
[LMP22]	LSB	$C_{\text{ext}} \leq (W+1) \cdot 2^{2E}$	$W \cdot \sigma_{\text{br}}^2$	$\frac{2}{\log_2(p)}$	✗
[KS24]	LSB	$C_{\text{ext}} \leq W \cdot 2^{2E}$	$\mathcal{O}(p^3) \cdot \sigma_{\text{br}}^2$	$\frac{2}{\log_2(p)}$	✗
Our work	MSB	$C_{\text{ext}} \leq (W+2) \cdot 2^{2E}$	σ_{br}^2	$\frac{1}{\log_2(p)-2}$	✓

as neural networks, that do not require exact computations. As the state-of-the-art starts with the LSBs, efficient approximate decomposition is not feasible with those techniques.

5 Application: Summation of a large number of ciphertexts

As previously discussed, the summation of a larger number of ciphertexts is a crucial operation in various TFHE applications. In this section, we will demonstrate how we can leverage our DirtyMSB technique to speed-up this process. For instance, we will apply our technique to parameter sets with a plaintext space $p = 32$, which corresponds to 5 bits. One of these bits is the padding bit, which is typically kept 0. This plaintext size is widely used in various TFHE applications and has been shown by Bergerat et al. [BBB+23] to provide the optimal trade-off between precision and accuracy. For larger integers, we employ the radix-based integer representation technique with the message-carry representation (2-bit carry, 2-bit message), as explained in Subsection 2.4.

To optimise the summation, we propose to use a larger plaintext space p_{ext} and denote it with hat notation: $\hat{ct}$. We show that we can use this to add a larger number of ciphertexts together in a three-step procedure:

1. Convert input ciphertexts to the extended-plaintext format using a PBS.
2. Combine the extended-plaintext ciphertexts using traditional LWE additions or subtractions.
3. Use the DirtyMSB procedure to perform digit decomposition and convert the extended-plaintext ciphertext back into multiple standard LWE ciphertexts.

5.1 The three-step approach

Conversion to extended-plaintext: This conversion can be performed by executing an identity bootstrap (i.e., the LUT computes $f(x) = x$), with the difference that the LUT outputs are scaled by a factor of Δ_{ext} instead of Δ. In practice, for an unencrypted LUT, this means that all coefficients in the LUT are shifted down by $\log_2(\Delta/\Delta_{\text{ext}})$ bits. In most cases, this procedure can be amortised in earlier computations by already using the lower scaling factor Δ_{ext} in the previous final PBS step, avoiding the need for an additional conversion step.

Summation: As the ciphertexts are still valid LWE encryptions, we can perform additions or subtractions as before, by executing the operation coefficient-wise. It is important to keep the padding bit clean to ensure correct extractions, which limits the total number of additions that can be performed. Alternatively, we could allow the padding bit to be used, which would enable more additions but require us to employ the WoP-PBS technique of

Chillotti et al. [CLOT21] for the first digit extraction. We leave how to efficiently combine this technique with our DirtyMSB approach as interesting future work.

Digit Decomposition: In this section, we modify the previously explained general DirtyMSB procedure to ensure that the output ciphertexts can be used in future computations. Depending on the computations, there are two distinct approaches. If the next computation requires regular plaintext ciphertexts, we convert a single extended-plaintext ciphertext into a list of regular plaintext ciphertexts. This conversion will be explained in more detail in Subsection 5.3. On the other hand, if the next computation requires extended-plaintext ciphertexts, we convert a single ciphertext into a list of new extended-plaintext ciphertexts, which will be explained in more detail in Subsection 5.4. The primary objective of both approaches is to generate multiple ciphertexts with an empty carry space, aligned with the message space blocks as in Equation 2.

5.2 Optimal Extended Plaintext Size

In Subsection 4.4, we demonstrated that bootstrap latency varies depending on the value of C_{ext}. This is because the parameter sets must be adjusted to ensure that the fault probability remains smaller than 2^{-64}. Later, in Equation 9, we also showed that C_{ext} depends on the initial input noise multiplier W and the number of extra bits in an extended-plaintext encryption, E. In the context of the summation operation, the initial noise multiplier W depends on the value E and the original number of plaintext bits in a ciphertext (excluding the padding bit), B. With increasing E and B, more summations can be performed before the plaintext space fills up, and consequently, the noise multiplier W will be larger. By assuming that the elements added together in our summation operation are independent from each other, we find that the noise grows linearly with the number of additions performed.

In the typical TFHE message-carry framework, where the number of message bits equal the number of carry bits, the initial input values before summation can be at most $\sqrt{p/2}-1 = 2^{B/2}-1$. This means that we can at most add together $\left\lfloor \frac{2^{E+B}}{2^{B/2}-1} \right\rfloor$ ciphertexts before the entire extended-plaintext space may be full, and we have to perform digit decomposition. Filling in this bound in W in Equation 9 gives us:

$$\begin{aligned} C_{\text{ext}} &\leq \left\lfloor \frac{2^{E+B}}{2^{B/2}-1} \right\rfloor \cdot 2^{2E} + 2 \cdot 2^{2E} \\ &= \mathcal{O}(2^{3E}). \end{aligned}$$

This results in a maximal noise variance on the order of $\sigma^2_{\text{ext}} = \mathcal{O}(2^{3E}) \cdot \sigma^2_{\text{br}}$. In other words, increasing the plaintext size by one bit roughly doubles the number of additions that can be performed W, but increases the noise variance of σ^2_{ext} by a factor of 8.

Table 2 provides an overview of the minimum C_{ext} values in relation to the number of bits added to the extended-plaintext ciphertext E, with $B=4$. Additionally, it shows the corresponding bootstrapping latency τ_{BS} achieved using a single thread. To make a fair comparison between the different options, we introduce a figure-of-merit that quantifies the number of ciphertexts that can be removed per millisecond: $\frac{W-k}{k \cdot \tau_{BS}}$. This is derived from the fact that for each option, we can combine W ciphertexts in k iterations, where each iteration generates a new ciphertext and requires a single bootstrap operation with a latency of τ_{BS}.

For the following application, we use $E=2$ (and consequently, the number of total plaintext bits is $E+B=6$). This approach maximises the number of ciphertexts reduced per second while minimising the increase in bootstrapping latency. Although the parameter set with $E=3$ may enable more efficient additions, it would also increase the cost of all other future bootstrapping operations.

Table 2: The minimum required C_{ext} and the resulting ciphertext removal rate (ciphertexts per second), plotted against E, the number of extra bits. Here, W indicates the number of ciphertexts that can be added together using the extended-plaintext, and k signifies the minimum number of bootstraps needed for the digit decomposition. The bootstrapping latency, τ_{BS}, was measured using an optimised parameter set for 128-bit security and an error probability of 2^{-64} with a single thread in the TFHE-rs library.

B	E	W	k	C_{ext}	$\tau_{BS}\ [ms]$	$\frac{W-k}{k\cdot\tau_{BS}}[ct/ms]$
4	0	5	1	5	10.2	392
4	1	10	2	44	10.3	388
4	2	21	2	352	10.4	913
4	3	42	2	2752	16.6	1205

5.3 6-bit to 4-bit Carry-Clean Extraction

As previously explained, depending on the next computation, we must decompose an extended-plaintext ciphertext into either a list of carry-free regular plaintext ciphertexts or a new list of extended-plaintext ciphertexts. The specific conversion process varies between these two scenarios. First, we look at the conversion into a list of regular plaintext ciphertexts. The objective is to split a 6-bit input ciphertext into multiple ciphertexts, each aligned with either $(a\|b)$, $(c\|d)$, or $(f\|g)$, corresponding to the message space blocks from Equation 2. A straightforward application of algorithm 1 with $T=[2,3,3]$ (as shown in Example 4.1) results in 3 ciphertexts. The most significant ciphertext has clean carries, while the other two are in the range [0,7] and thus have one carry bit, as depicted on the left side of Figure 6. This procedure utilises 3 PBS with an additional 2 PBS (using manyLUT) required to clean the carries, resulting in a total of 5 PBS.

On the right side, we propose a different method to achieve clean conversion in 3 PBS creating four different ciphertext. First, one performs two DirtyMSB extractions on the input $\hat{ct}$ to extract both a dirty version of $(a\|b)$ in ct_{ab^*} and $(c\|d)$ in ct_{cd^*}. Note that this is done in parallel and that we do not first subtract ct_{ab^*} from the input ciphertext before extracting $(c\|d)$. The advantage of this approach is that both ct_{ab^*} and ct_{cd^*} have clean carries.

Second, to extract the remaining bits, we subtract $4\cdot ct_{ab^*}+ct_{cd^*}$ from the extended-plaintext ciphertext and perform a 2-bit right shift. This allows us to use a double lookup (using manyLUT) to extract both ct_{fg} and the carry bit δ, aligned with $(c\|d)$. This means that our adapted extraction procedure results in four regular plaintext ciphertexts, $(a\|b)$, $(c\|d)$, $(0\|\delta)$, and $(f\|g)$, as shown in Figure 6. Note that both methods are identical in terms of noise, where one can use Equation 6 with $W=21$ to get:

$$C_{\text{ext}}=16\cdot(21+2^4+1)=608.$$

5.4 6-bit to 6-bit plaintext Carry-Clean Extraction

If future operations once again require extended-plaintext ciphertexts, we can slightly modify the conversion so that the resulting ciphertexts are all in that format. The only necessary change is to replace the Δ used in the PBS function with Δ_{ext}. However, since the extracted ciphertexts are in extended plaintext format, we must multiply them with an additional factor $E^2=4$ before subtracting them from the original ciphertext. This extra shift causes parts of the noise variance to be amplified with a factor $E^{2^2}=4^2$, resulting in a C_{ext} of:

$$C_{\text{ext}}=16\cdot(21+4^2\cdot(2^4+1))=4688.$$

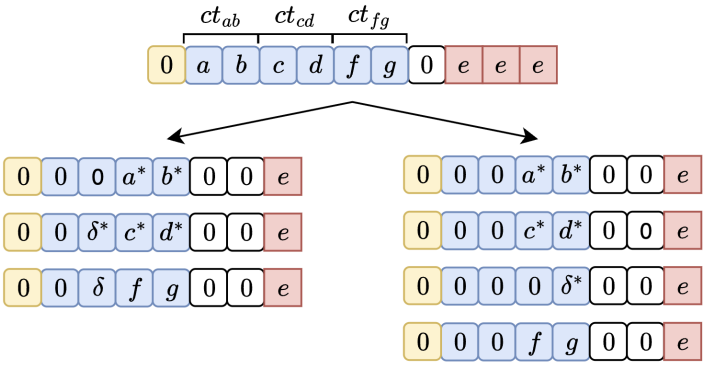

Figure 6: 6-bit to 4-bit conversion using the non-carry-clean and carry-clean approach

Unfortunately, this C_{ext} necessitates a less efficient parameter set, counteracting the benefit of using the extended plaintext approach. Therefore, we suggest adapting the method and using one additional PBS operation that extracts a second ciphertext containing $(a\|b)$ in the MSBs (using $16\cdot\Delta_{\text{ext}}$, instead of Δ_{ext}). We use this sample to subtract the relevant bits out of the initial ciphertext, which results in $C_{\text{ext}} = 16\cdot(21+2^4+1) = 608$, which is the same value we found for the 6-bit to 4-bit conversion. However, we now require four PBS operations instead of three.

6 Results

In this section, we apply our DirtyMSB technique to the vector addition operation and scalar multiplication and demonstrate the resulting improvements. To measure efficiency, we will count the number of PBS required for the operation and measure the latency. We assume that the conversion of inputs ciphertexts to extended plaintext format is already performed in the previous operation. As explained earlier, this is a reasonable assumption as it only requires the last bootstrap to use the different scaling factor Δ_{ext} instead of Δ.

As explained earlier, we stick to 6-bit extended-plaintext ciphertexts. This choice is driven by the fact that larger ciphertext spaces would lead to increased PBS costs for other operations, as was illustrated in Table 2. Consequently, we will employ our carry-clean 6-bit extraction techniques and integrate them into the vector addition operation implemented in the TFHE-rs library [Zam22], and compare this approach to the conventional message-carry extraction techniques. To ensure a fair comparison, we select the most optimal parameter set for each work, considering the maximum possible value of C_{ext}.

6.1 Vector Addition

To benchmark the efficiency of our digit decomposition in a practical application, we consider vector addition, where multiple encrypted integers are summed. This operation is not only useful on its own but also serves as a fundamental building block for other operations. In the next section, we will demonstrate how this speedup impacts scalar multiplication.

As a reference, using our approach described in the previous section, we can add 21 ciphertexts before the carry bits are saturated. At this point, we must extract them into four separate ciphertexts using either three or four bootstraps, depending on the format of the extracted ciphertexts. For large-scale vector additions, we repeatedly perform 6-bit to 6-bit extraction, requiring four bootstraps until the number of ciphertexts remaining in the vector is less than 21. Then, we add them together and use the 6-bit to 4-bit extraction procedure. This means that at the end of the vector addition, the ciphertext once again encrypts a regular-sized

Table 3: Comparison of the number of bootstrap operations and corresponding latency for the vector addition operation using the standard approach in the TFHE-rs library versus our method. The latency was measured using 8 threads on an AMD EPYC 9174F 16-core processor.

#Elements /Bitsize	#PBS Operations			Latency		
	TFHE-rs	Ours	Improv	TFHE-rs	Ours	Improv
10/8-bit	14	11	×1.27	29ms	33ms	×0.90
20/8-bit	33	11	×3.00	68ms	34ms	×1.98
50/8-bit	99	37	×2.68	193ms	92ms	×2.09
10/16-bit	30	23	×1.30	57ms	58ms	×0.98
20/16-bit	73	23	×3.17	142ms	58ms	×2.44
50/16-bit	219	81	×2.70	405ms	175ms	×2.32
200/16-bit	937	330	×2.84	1.685s	651ms	×2.59
1000/16-bit	4762	1649	×2.89	8.676s	3.150s	×2.75
10/32-bit	62	47	×1.32	115ms	104ms	×1.10
20/32-bit	153	47	×3.26	274ms	106ms	×2.59
50/32-bit	459	169	×2.72	824ms	343ms	×2.4
200/32-bit	1977	706	×2.80	3.547s	1.347s	×2.63
1000/32-bit	10074	3529	×2.85	18.252s	6.642s	×2.75

plaintext and can be directly used for further computation.

Theoretically, this implies that we can compress 21 ciphertexts to four ciphertexts in 4 bootstraps resulting in a theoretical PBS efficiency of $(21-4)/4=4.25$. Compared to the 1.5 PBS efficiency ($(5-2)/2=1.5$) of the approach used in the TFHE-rs library of Zama [Zam22], this gives us a theoretical improvement of ×2.83.

In Table 3, an overview is provided for both the number of bootstrap operations and their corresponding latency for various vector and bit sizes. The tests were performed on an AMD EPYC 9174F 16-core processor with a maximum of 8 threads. Note that, since TFHE-rs by Zama employs the standard 4-bit ciphertexts, it has a maximum $C_{\text{ext}}=5$. Therefore, it can utilise a slightly more efficient parameter set compared to our 6-bit scenario, as shown in Table 2. Our digit decomposition approach achieves latency improvements of up to ×2.75. It is worth mentioning that even for vector sizes smaller than 21, the digit decomposition approach approximately achieves the same performance as the original approach.

6.2 Multi-Scalar Multiplication

In recent work by Pottier et al. [PDdRV25] on the optimization of the scalar multiplication, the well-known Pippenger method from the domain of elliptic curves was adapted to the TFHE case, resulting in more efficient operations. The scalar multiplication eventually comes down to a precomputation stage and a vector addition stage. While Pottier et al. reduced the number of elements in the vector, vector addition still remains the bottleneck in the scalar multiplication algorithm. As discussed in the previous section, our digit decomposition approach enables the vector addition operation to be up to ×2.75 more efficient. By combining our approach with the one of Pottier et al., we achieved a total improvement up to ×16.74 over TFHE-rs, an additional improvement of ×2.23 over Pottier et al. as shown in Table 4.

Table 4: Comparison between recent works for the multi-scalar multiplication of 16-bit scalars and ciphertext for different number of elements. The latency was measured using 16 threads on an AMD EPYC 9174F 16-core processor.

#Elements	TFHE-rs	Pottier et al. [PDdRV25]	This work + [PDdRV25]	Improv over TFHE-rs	Improv over [PDdRV25]
2400	111.3s	16.6s	8.1s	×13.73	×2.04
4096	186.0s	26.9s	11.84s	×15.71	×2.27
9216	421.6s	56.1s	25.19s	×16.74	×2.23

7 Conclusion

In this paper, we present a new digit decomposition technique that proceeds from the most significant bit to the least significant bit. We show that our technique is faster than the state-of-the-art for larger plaintexts ($\log_2(p) \geq 4$). Additionally, our technique reduces the output noise and returns 'bootstrap clean' ciphertexts, while allowing approximate digit decomposition at lower cost. We use this new digit decomposition technique to optimise the summation of a large number of ciphertexts in the context of the message-carry framework, achieving a speedup of ×2.79 over state-of-the-art vector addition techniques. Integration of this technique in scalar multiplication shows a speedup of up to ×2.27 over previous works. Interesting future work includes efficiently combining the dirty MSB approach with the WoP-PBS algorithm from Chillotti et al. [CLOT21], resulting in a higher number of bits extracted per iteration at the cost of more output noise. This presents an additional trade-off that can be modified to different applications.

Acknowledgments This work was supported in part by the Horizon 2020 ERC Advanced Grant (101020005 Belfort) and the CyberSecurity Research Flanders with reference number VOEWICS02. Xander Pottier is funded by FWO (Research Foundation – Flanders) as Strategic Basic (SB) PhD fellow (project number 1S93126N).

A TFHE-Parameter sets

Table 5: TFHE Parameter sets depending on the range of C_{ext}, ensuring 128-bit security and an error probability of 2^{-64}. Latency benchmarks were measured using the TFHE-rs library with the mean compensation optimisation of de Ruijter et al. [dRDV25] using 1 thread on an AMD EPYC 9174F 16-core processor.

$C_{\text{ext, min/max}}$	Latency	N	k	n	l_{BS}	β_{BS}	l_{KS}	β_{KS}	σ^2_{LWE}	σ^2_{GLWE}
1 − 42	10.181 ms	2048	1	752	1	2^{23}	7	2^2	$2^{-16.71}$	$2^{-50.29}$
43 − 262	10.330 ms	2048	1	758	1	2^{23}	7	2^2	$2^{-16.86}$	$2^{-50.29}$
263 − 441	10.432 ms	2048	1	763	1	2^{23}	7	2^2	$2^{-17.0}$	$2^{-50.29}$
442 − 582	10.424 ms	2048	1	769	1	2^{23}	7	2^2	$2^{-17.14}$	$2^{-50.29}$
583 − 695	10.609 ms	2048	1	775	1	2^{23}	7	2^2	$2^{-17.29}$	$2^{-50.29}$
696 − 786	10.630 ms	2048	1	780	1	2^{23}	7	2^2	$2^{-17.43}$	$2^{-50.29}$
787 − 856	10.691 ms	2048	1	786	1	2^{23}	7	2^2	$2^{-17.57}$	$2^{-50.29}$
857 − 910	10.796 ms	2048	1	792	1	2^{23}	7	2^2	$2^{-17.71}$	$2^{-50.29}$
911 − 954	10.835 ms	2048	1	797	1	2^{23}	7	2^2	$2^{-17.86}$	$2^{-50.29}$
955 − 986	10.971 ms	2048	1	803	1	2^{23}	7	2^2	$2^{-18.0}$	$2^{-50.29}$
987 − 1011	11.037 ms	2048	1	808	1	2^{23}	7	2^2	$2^{-18.14}$	$2^{-50.29}$
1012 − 1028	11.184 ms	2048	1	814	1	2^{23}	7	2^2	$2^{-18.29}$	$2^{-50.29}$
1029 − 1039	11.247 ms	2048	1	820	1	2^{23}	7	2^2	$2^{-18.43}$	$2^{-50.29}$
1040 − 1048	11.306 ms	2048	1	825	1	2^{23}	7	2^2	$2^{-18.57}$	$2^{-50.29}$
1049 − 1065	11.533 ms	2048	1	846	1	2^{23}	6	2^3	$2^{-19.14}$	$2^{-50.29}$
1066 − 1038651	16.624 ms	2048	1	752	2	2^{15}	7	2^2	$2^{-16.71}$	$2^{-50.29}$

References

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 9(3):169–203, 2015.

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

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3):13:1–13:36, 2014.

[BIP+22] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. FINAL: Faster FHE instantiated with NTRU and LWE. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 188–215. Springer, Cham, December 2022.

[BMMP18] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 483–512. Springer, Cham, August 2018.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 3–33. Springer, Berlin, Heidelberg, December 2016.

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

[CH18] Hao Chen and Kyoohyung Han. Homomorphic lower digits removal and improved FHE bootstrapping. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 315–337. Springer, Cham, April / May 2018.

[CIM19] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. New techniques for multi-value input homomorphic evaluation and applications. In Mitsuru Matsui, editor, *CT-RSA 2019*, volume 11405 of *LNCS*, pages 106–126. Springer, Cham, March 2019.

[CJP21] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander Schwarzmann, editors, *Cyber Security Cryptography and Machine Learning*, pages 1–19, Cham, 2021. Springer International Publishing.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 409–437. Springer, Cham, December 2017.

[CLOT21] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 670–699. Springer, Cham, December 2021.

[CZB+22] Pierre-Emmanuel Clet, Martin Zuber, Aymen Boudguiga, Renaud Sirdey, and Cédric Gouy-Pailler. Putting up the swiss army knife of homomorphic calculations by means of TFHE functional bootstrapping. Cryptology ePrint Archive, Report 2022/149, 2022.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 617–640. Springer, Berlin, Heidelberg, April 2015.

[dRDV25] Thomas de Ruijter, Jan-Pieter D'Anvers, and Ingrid Verbauwhede. Don't be mean: Reducing approximation noise in TFHE through mean compensation. Cryptology ePrint Archive, Paper 2025/809, 2025.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012.

[GBA21] Antonio Guimarães, Edson Borin, and Diego F. Aranha. Revisiting the functional bootstrap in TFHE. *IACR TCHES*, 2021(2):229–253, 2021.

[GGP+23] Ravit Geva, Alexander Gusev, Yuriy Polyakov, Lior Liram, Oded Rosolio, Andreea Alexandru, Nicholas Genise, Marcelo Blatt, Zohar Duchin, Barliz Waissengrin, Dan Mirelman, Felix Bukstein, Deborah T. Blumenthal, Ido Wolf, Sharon Pelles-Avraham, Tali Schaffer, Lee A. Lavi, Daniele Micciancio, Vinod Vaikuntanathan, Ahmad Al Badawi, and Shafi Goldwasser. Collaborative privacy-preserving analysis of oncological data using multiparty homomorphic encryption. Cryptology ePrint Archive, Report 2023/1203, 2023.

[GV25] Robin Geelen and Frederik Vercauteren. Fully homomorphic encryption for cyclotomic prime moduli. In *Advances in Cryptology – EUROCRYPT 2025: 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Madrid, Spain, May 4–8, 2025, Proceedings, Part III*, page 366–397, Berlin, Heidelberg, 2025. Springer-Verlag.

[HS15] Shai Halevi and Victor Shoup. Bootstrapping for HElib. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 641–670. Springer, Berlin, Heidelberg, April 2015.

[ISZ19] Malika Izabachène, Renaud Sirdey, and Martin Zuber. Practical fully homomorphic encryption for fully masked neural networks. In Yi Mu, Robert H. Deng, and Xinyi Huang, editors, *CANS 19*, volume 11829 of *LNCS*, pages 24–36. Springer, Cham, October 2019.

[KS23] Kamil Kluczniak and Leonard Schild. FDFB: Full domain functional bootstrapping towards practical fully homomorphic encryption. *IACR TCHES*, 2023(1):501–537, 2023.

[KS24] Kamil Kluczniak and Leonard Schild. FDFB^2: Functional bootstrapping via sparse polynomial multiplication. Cryptology ePrint Archive, Report 2024/1376, 2024.

[LMP22] Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. Large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 130–160. Springer, Cham, December 2022.

[LY23] KangHoon Lee and Ji Won Yoon. Discretization error reduction for high precision torus fully homomorphic encryption. In Alexandra Boldyreva and

Vladimir Kolesnikov, editors, *PKC 2023, Part II*, volume 13941 of *LNCS*, pages 33–62. Springer, Cham, May 2023.

[Max21] Nick J Maxwell. Innovation and discussion paper: Case studies of the use of privacy preserving analysis to tackle financial crime. *Royal United Services Institute for Defence and Security Studies, FFIS*, 2021.

[PDdRV25] Xander Pottier, Jan-Pieter D'Anvers, Thomas de Ruijter, and Ingrid Verbauwhede. SMOOTHIE: (multi-)scalar multiplication optimizations on TFHE. Cryptology ePrint Archive, Paper 2025/1267, 2025.

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

[WBS+25] Ruida Wang, Jikang Bai, Xuan Shen, Xianhui Lu, Zhihao Li, Binwu Xiang, Zhiwei Wang, Hongyu Wang, Lutan Zhao, Kunpeng Wang, and Rui Hou. Tetris: Versatile TFHE LUT and its application to FHE instruction set architecture. Cryptology ePrint Archive, Paper 2025/1623, 2025.

[WHS+25] Ruida Wang, Jincheol Ha, Xuan Shen, Xianhui Lu, Chunling Chen, Kunpeng Wang, and Jooyoung Lee. Refined tfhe leveled homomorphic evaluation and its application. *Proceedings of the 2025 ACM SIGSAC conference on COmputer and Communications Security*, 2025.

[WWL+24] Ruida Wang, Yundi Wen, Zhihao Li, Xianhui Lu, Benqiang Wei, Kun Liu, and Kunpeng Wang. Circuit bootstrapping: Faster and smaller. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part II*, volume 14652 of *LNCS*, pages 342–372. Springer, Cham, May 2024.

[YXS+21] Zhaomin Yang, Xiang Xie, Huajie Shen, Shiying Chen, and Jun Zhou. TOTA: Fully homomorphic encryption with smaller parameters and stronger security. Cryptology ePrint Archive, Report 2021/1347, 2021.

[Zam22] Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. https://github.com/zama-ai/tfhe-rs.

[ZLS+19] Yandong Zheng, Rongxing Lu, Jun Shao, Yonggang Zhang, and Hui Zhu. Efficient and privacy-preserving edit distance query over encrypted genomic data. In *2019 11th International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–6. IEEE, 2019.