Designing a General-Purpose 8-bit (T)FHE Processor Abstraction*

Daphné Trama, Pierre-Emmanuel Clet, Aymen Boudguiga, Renaud Sirdey, and Nicolas Ye

Université Paris-Saclay, CEA-List, Palaiseau, France
{name.lastname}@cea.fr

Abstract

Making the most of TFHE programmable bootstrapping to evaluate functions or operators otherwise challenging to perform with only the native addition and multiplication of the scheme is a very active line of research. In this paper, we systematize this approach and apply it to build an 8-bit FHE processor abstraction, i.e., a software entity that works over FHE-encrypted 8-bit data and presents itself to the programmer by means of a conventional-looking assembly instruction set. In doing so, we provide several homomorphic LookUp Table (LUT) dereferencing operators based on variants of the tree-based method and show that they are the most efficient option for manipulating encryptions of 8-bit data (optimally represented as two basis 16 digits). We then systematically apply this approach over a set of around 50 instructions, including, notably, conditional assignments, divisions, or fixed-point arithmetic operations. We further test the approach on several simple algorithms, including the execution of a neuron with a sigmoid activation function with 16-bit precision. We conclude the paper by comparing our work to the FHE compilers available in the state of the art. Finally, this work reveals that a very limited set of functional bootstrapping patterns is versatile and efficient enough to achieve general-purpose FHE computations beyond the boolean circuit approach. As such, these patterns may be an appropriate target for further works on advanced software optimizations or hardware implementations.
Keywords — Fully Homomorphic Encryption · TFHE · Programmable Bootstrapping · General Computations

1 Introduction

The key idea behind homomorphic encryption is to be able to perform *any* calculation directly over ciphertexts. In the early years of FHE, the hope was to achieve this goal by executing boolean circuits over ciphertexts encoding binary messages with both XOR and AND (homomorphic) gates. Although this computing model is universal, it also leads to many efficiency bottlenecks: for example, to merely perform a simple multiplication over $\mathbb{Z}_t$ ($t >> 2$), one has to perform many boolean operations, and even more so for more complex

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

operations such as divisions. Because of this, works on FHE have progressively departed from this paradigm to focus on running arithmetic circuits over polynomial rings with a plaintext modulus much larger than 2. In doing so, FHE efficiency has greatly improved, allowing it to address concrete applications, for example, in the field of Machine Learning, with reasonable latencies and overheads. However, this latter approach comes with difficult challenges for applications in need of zero testing or other non-linear functions[1]. At the other end of the spectrum stands TFHE. On the downside, TFHE is intrinsically an LWE scheme, meaning that it offers no batching (except for additions) and only allows for small plaintext moduli (e.g., less than 32). On the bright side, TFHE has the most efficient bootstrapping procedure, which is furthermore "programmable". Indeed, TFHE bootstrapping refreshes ciphertext noise essentially by interpreting the input ciphertext as an encrypted index for dereferencing a cleartext table encoding the identity function with some redundancy. When the identity function is replaced by another function $f : \mathbb{Z}_t \longrightarrow \mathbb{Z}_t$, the bootstrapping operation evaluates f "for free". As such, compared to the raw boolean-circuit approach, TFHE offers a toolbox to mitigate its efficiency bottlenecks by supporting a non-binary (albeit still small) plaintext domain $\mathbb{Z}_{2^k}$, thus allowing to factor the evaluation of k-bit to k-bit boolean circuits in single bootstrapping operations.

Natural questions to explore are then the following. Can we build on the TFHE functional bootstrapping toolbox to achieve universal encrypted domain computations beyond the boolean circuit approach? And at which computational cost? Can this be achieved from a restricted set of patterns based only on functional bootstrapping, hence with a homogeneous algorithmic structure? In this paper, we give a first answer to these questions by designing and implementing a general-purpose 8-bit FHE processor abstraction working over encryptions of bytes represented by pairs of TFHE ciphertexts encoding their most and least significant nibbles[2]. As one may intuit, however, the resulting instruction set is quite different from that of a usual processor. Many instructions cannot be straightforwardly performed in the encrypted domain, and new instructions must be provided to work around these limitations. For example, the lack of conditional branching instructions (an FHE "processor" can evaluate any condition but *cannot* access the resulting encrypted boolean to branch) has to be worked around by providing a set of conditional assignment instructions.

Our approach heavily relies on TFHE programmable bootstrapping, one of the first uses of which was calculating the sign function [BMMP18], notably for evaluating a new class of strongly discretized neural networks over FHE encrypted inputs. However, this programmable bootstrapping can only natively be used on a single input ciphertext. Thus, to overcome such limitations, we need bootstrapping composition techniques to homomorphically evaluate functions on larger data types represented by several encryptions of their basis B digits. A number of such methods have been proposed and investigated in the literature in recent years, such as the tree-based and chain-based methods [GBA21], the WoP-PBS [BBB+23] and the p-encoding method [BPR24]. We first review these methods and select the most appropriate one along with the most suitable basis B to design a small set of generic operators for dereferencing one or more LookUp Table (LUT) with 256 entries in $\mathbb{Z}_B$ using an encrypted index. We then systematically use these operators to build our set of 8-bit instructions.

[1] Despite several attempts using bivariate polynomial optimizations [IZ21] or polynomial approximations [LLKN21, CKK19] for implementing comparisons and zero-testing with schemes such as BFV/BGV or CKKS.

[2] "Nibble" is the cute name for 4-bits entities

1.1 Summary of Contributions

This paper's contributions are as follow:

- We show that the most optimized approach for manipulating TFHE encryptions of 8-bit messages is achieved by using the tree-based functional bootstrapping method regardless of the decomposition basis $B > 2$. We further show that 8-bit messages are optimally represented and manipulated as two basis 16 digit encryptions. This conclusion is valid for all operations except bitwise ones for which basis 4 is optimal.
- We define a set of functional bootstrapping tools and optimal parameters to manipulate encryptions of 8-bit data by means of LUT dereferencing with TFHE ciphertexts. We designed this toolboox such that blind rotations and keyswitches (the most costly operations within TFHE bootstrapping) can be factored as much as possible to improve efficiency. By analogy to a real microprocessor, this can be seen as the micro-code level of a processor abstraction.
- We then define a complete set of over 50 instructions suitable for working with TFHE encryptions of 8-bit data, including FHE-specific instructions as well as advanced operators such as conditional assignment, division, or even fixed-point arithmetic operations among many others. For each of these instructions, we provide strategies to efficiently instantiate them using our LUT-dereferencing building blocks. To the best of our knowledge, we present the first ever concrete implementation of the Euclidean and decimal division operators over FHE not relying on the boolean circuit approach.
- We test our approach over several higher-level simple algorithms (sorting, average computation, finding the minimum or maximum of an array, ...) and provide extensive timing experiments. To the best of our knowledge, we provide the first FHE instantiation of a fixed-precision sigmoid function over 16 bits leading to the FHE instantiation of *standard* neurons which can be seamlessly chained to enable the evaluation of larger (possibilty recurrent) neural networks over encrypted data.
- We compare our approach to the state-of-the-art FHE compilers and related approaches, including Cingulata [CDS15a], E3 [EOH+18], Concrete [Zam22] and Juliet [GMT24]. We demonstrate performance improvements between 60% and 99% on most algorithms (except those inducing very small Boolean circuits).
- As we essentially define the first FHE-oriented ISA, this work is a significant first step towards defining virtual FHE machine architectures to bridge the gap with standard existing compilers and benefiting from their powerful code optimizations engines to both express more complex programs and improve their execution performances over FHE.
- Lastly, as a matter of perspectives, we carefully analyze the computational hotspots in the approach, providing cleanly defined candidate kernels of increasing complexity for low-level optimizations or even hardware acceleration.

1.2 Paper Organization

This paper is organized as follows: Section 2 reviews the related works and Section 3 recalls the basics of the TFHE cryptosystem and gives the necessary details on the tree-based method for bootstrapping with multi-input ciphertexts as well as its optimization with multi-value bootstrapping. Then, Section 4 details the rationale for selecting the functional bootstrapping technique and its associated parameters. Sections 6 and 7 (unitary timings)

subsequently focus on our instruction set, which is then used in Section 8 to implement several algorithms. We then compare our ISA-based approach to the FHE compilers in the state of the art in Section 9. Finally, Section 10 concludes this paper with some perspectives. We also provide a number of appendices: Appendix A provides an exhaustive table giving the description and the unitary timings of all of our operators, Appendix B gives further details on the optimized implementation for all the instructions we propose and Appendix C includes more background details on TFHE.

2 Related Work

2.1 FHE Virtual Processors

To the best of our knowledge, previous attempts at building FHE-based virtual processors are pretty scarce and, for the most part, date back to the first few years after Gentry's breakthrough. By *virtual processor* or *processor abstraction*, we mean a software entity that works over FHE-encrypted data and presents itself to the programmer by means of a conventional-looking assembly instruction set. Perhaps the first attempt is that of Brenner et al. [BPS][3], which was based on the Smart-Vercauteren scheme [SV10]. This work proposes an abstract processor that executes an *encrypted program* (over encrypted data). The processor has a minimal instruction set containing only bitwise logical operations as well as load/store (with encrypted addresses, hence with an access complexity linear in memory size) and three branching instructions. Each (encrypted) instruction is fetched from the encrypted memory and then homomorphically interpreted (at an extra cost equivalent to explicitly running all instructions in the set). Being more than twelve years old, from an experimental point of view, this latter work is obsolete. Still, our approach departs significantly from it because we run *public* programs over encrypted data, i.e., the stream of instructions is *not* encrypted. The main consequence is that we restore constant-time memory access (because all the addresses are public) but cannot perform any branching (conditioned on encrypted values). Branching then has to be emulated using explicit conditional assignments at an extra cost equivalent to that of explicitly running all branches. In that sense, our programming model somewhat resembles the "constant time programming" model often used in embedded computing [ABB+16]. Also, we can then afford to have a much more complete instruction set, which is tailored to the capabilities of our modern functional bootstrapping toolbox. Another attempt is that of [FSF+13], which considers a richer set of operators (rather than explicit instructions) and is boolean-circuit oriented. From an experimental viewpoint, this latter work is also too old not to be obsolete. Other works include experiments at building a one-instruction set processor abstraction working over FHE-encrypted data [TM14, TM13, CS19], an approach which also achieves Turing completeness but leads to even worse blow-ups in the number of instructions than the boolean circuit one. A more recent attempt at supporting a subset of the ARM (v8) instruction set over TFHE is given in [GN20]. This approach has two main drawbacks. First, it uses TFHE only in gate-bootstrapping mode and, as such, does not work over a larger plaintext space as we do with functional bootstrapping techniques. Second, it handles conditional branching in a client-aided fashion with the consequence of granting the FHE processor access to a decryption oracle. This is likely to induce vulnerabilities in realistic deployment scenarios since TFHE is trivially insecure against a CCA(1) adversary. By opposition, we "handle"

[3] `github.com/hcrypt-project`.

branching in a non-interactive way via conditional assignment instructions (but at the extra cost of running all branches). Lastly, a few works [IMP18, CGRS14] propose to extend the instruction set of existing processors with a small set of additional instructions for driving FPGA-implementations of FHE operations (with [IMP18] also handling branching in a client-aided fashion). On top of the above, there presently are many works on hardware implementation of FHE building blocks without any focus on instruction sets.

3 Preliminaries

3.1 Notations

Let $\mathcal{E} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ denote an FHE scheme with key space $\mathcal{K}$, plaintext domain $\mathcal{M}$ and ciphertext domain $\mathcal{C}$. For a message $m \in \mathcal{M}$, we denote $[\![m]\!] \subset \mathcal{C}$, the set of all its valid encryptions, which we sometimes refer to as the ciphertext class of m. Let $\mathcal{F}$ be the function domain of Eval i.e., $\mathsf{Eval} : \mathcal{F} \times \mathcal{C}^* \longrightarrow \mathcal{C}$ is such that for all $(\mathsf{ek}, \mathsf{dk}) \in \mathcal{K}$, all $f \in \mathcal{F}$ and all $m_1, \cdots, m_K \in \mathcal{M}^K$,

$$\mathsf{Eval}(f, \mathsf{Enc}(m_1), \cdots, \mathsf{Enc}(m_K))) \in [\![f(m_1, \cdots, m_K)]\!].$$

Unless otherwise stated, the (uppercase or lowercase) letter c always denotes a ciphertext. Other (uppercase or lowercase) letters denote plaintexts.

Let $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ be the real torus, that is to say, the additive group of real numbers modulo 1 ($\mathbb{R} \mod 1$). We further denote by $\mathbb{T}_N[X]^n$ the set of vectors of size n whose coefficients are polynomials of $\mathbb{T}[X] \mod (X^N + 1)$. N is usually a power of 2.

3.2 The TFHE Scheme

The TFHE scheme is a fully homomorphic encryption scheme [CGGI19] notably implemented in the TFHE library[4]. TFHE defines three structures to encrypt plaintexts, which we introduce below:

- TLWE sample: A pair $(a, b) \in \mathbb{T}^{n+1}$, where a is uniformly sampled from $\mathbb{T}^n$ and $b = \langle a, s \rangle + e$. The secret key s is uniformly sampled from $\mathbb{B}^n$, and the error $e \in \mathbb{T}$ is sampled from a Gaussian distribution with mean 0 and standard deviation σ.
- TRLWE sample: A pair $(a, b) \in \mathbb{T}_N[X]^{k+1}$, where a is uniformly sampled from $\mathbb{T}_N[X]^k$ and $b = \langle a, s \rangle + e$. The secret key s is uniformly sampled from $\mathbb{B}_N[X]^k$, the error $e \in \mathbb{T}_{\mathbb{N}}[\mathbb{X}]$ is a polynomial with random coefficients sampled from a Gaussian distribution with mean 0 and standard deviation σ. One usually chooses $k = 1$; therefore, a and b are polynomials.
- TRGSW sample: a vector of $(k+1)l$ TRLWE fresh samples.

Let $\mathcal{M}$ denote the discrete message space ($\mathcal{M} \in \mathbb{T}_N[X]$ or $\mathcal{M} \in \mathbb{T}$)[5]. To encrypt a message $m \in \mathcal{M}$, we add what is called a *noiseless trivial* ciphertext $(0, m)$ to a fresh encryption of 0. We denote by $c = (a, b) + (0, m) = (a, b + m) \in \mathrm{T(R)LWE}_s(m)$ the T(R)LWE encryption of m with key s. A message $m \in \mathbb{Z}[X]$ can also be encrypted in TRGSW samples by adding $m \cdot H$ to a TRGSW sample of 0, where H is a gadget decomposition matrix. As we will not

[4] tfhe.github.io/tfhe/

[5] In practice, we discretize the torus with respect to our plaintext modulus. For example, if we want to encrypt $m \in \mathbb{Z}_4 = \{0, 1, 2, 3\}$, we encode it in $\mathbb{T}$ as a value in $\mathcal{M} = \{0, 0.25, 0.5, 0.75\}$. As a slight abuse of notation, we use $\mathcal{M}$ to denote both the message space and its encodings on $\mathbb{T}$.

explicitly need such an operation in this paper, more details about TRGSW can be found in [CGGI19]. To decrypt a ciphertext c, we first calculate its phase $\phi(c) = b - \langle a, s \rangle = m + e$. Then, we need to remove the error, which is achieved by rounding the phase to the nearest valid value in $\mathcal{M}$. This procedure fails if the error exceeds half the distance between two consecutive elements of $\mathcal{M}$.

3.3 TFHE Bootstrapping and Programmable Bootstrapping

3.3.1 TFHE Bootstrapping

Bootstrapping is the operation that reduces the noise of a ciphertext, thus allowing further homomorphic calculations. It relies on three basic operations, which we briefly review in this section (refer to Section C.1 for more details). The first operation, BlindRotate, rotates a plaintext polynomial $testv$[6] by a TLWE encrypted index $c \in [\![m]\!]$. It returns a TRLWE encrypted polynomial of $testv \cdot X^{\phi(c)} \mod (X^N + 1)$, where $\phi(c)$ is the phase of c rescaled in $\mathbb{Z}_{2N}$. Then, one must apply the TLWESampleExtract, which extracts a coefficient from an encrypted TRLWE polynomial and converts it into a corresponding TLWE ciphertext. Finally, the PublicFunctionalKeyswitch enables the switching of keys and parameters. It is used to switch the extracted TLWE ciphertext to an encryption of the same message but with the initial key. In practice, the computation time of a TFHE bootstrapping depends mainly on the efficiency of the BlindRotate [CBSZ23]. *So, from now on, we will denote by* $\mathsf{N_{br}}$ *the number of* BlindRotate *required to evaluate a function on encrypted data.* Using $\mathsf{N_{br}}$ as a criterion simplifies comparing instructions implemented with the same set of TFHE parameters.

3.3.2 Programmable Bootstrapping

Bootstrapping involves doing an indirection in a table using an encrypted index while reducing noise. Indeed, if we set the coefficients of $testv$ to the results of the evaluation of a function f on elements of $\mathcal{M}$, performing the bootstrapping on this new $testv$ outputs $c' \in [\![f(m)]\!]$. That is to say, the bootstrapping gives an encryption of $f(m)$ without any additional cost and allows the evaluation of a LUT of f. We refer to this bootstrapping as *programmable* or *functional*. We note that the original bootstrapping (in [CGGI19]) is a particular case of programmable bootstrapping with f set to the identity function.

TFHE programmable bootstrapping is natively well-suited but limited to implementing LUTs of negacyclic functions[7] for two reasons. First, TFHE plaintext space is $\mathbb{T}$, where $[0, \frac{1}{2})$ corresponds to positive values and $[\frac{1}{2}, 1)$ to negative ones. So, if c is a TLWE encryption of a positive value, its phase $\phi(c)$ lies in $[0, \frac{1}{2})$, and it satisfies $\phi(c) \in [0, N)$ after rescaling to $\mathbb{Z}_{2N}$. Conversely, if c is a TLWE encryption of a negative value, its phase satisfies $\phi(c) \in [N, 2N)$ after rescaling to $\mathbb{Z}_{2N}$. Second, BlindRotate outputs an encryption of $testv$ mulitplied by $X^{\phi(c)} \mod (X^N + 1)$[8]. So, if $testv$ coefficients are set to the evaluation of a negacyclic function on the positive values of $\mathcal{M}$ (values in $\mathcal{M} \cap [0, \frac{1}{2})$), a bootstrapping with an input TLWE ciphertext c encrypting m returns either $f(m)$ if $m \in \mathcal{M} \cap [0, \frac{1}{2})$, or $-f(m - \frac{1}{2})$ if $m \in \mathcal{M} \cap [\frac{1}{2}, 1)$.

[6] We sometimes refer to this polynomial as the test polynomial or vector.

[7] Negacyclic functions are antiperiodic functions over $\mathbb{T}$ with period $\frac{1}{2}$, satisfying $f(x) = -f(x + \frac{1}{2})$.

[8] We remind that $\forall \alpha \in [0, N), X^{\alpha+N} = -X^{\alpha} \mod (X^N + 1)$.

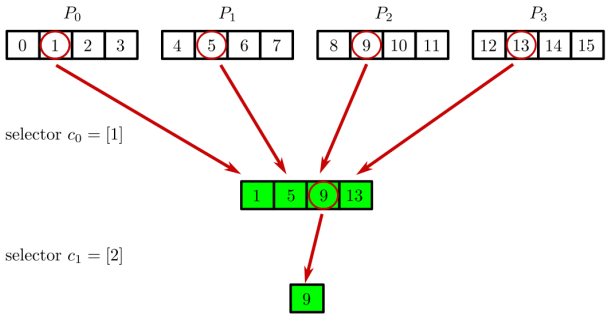

Figure 1: Illustration of the tree-based method on the identity function with decomposition in basis $B = 4$. The message is $m = 9 = 1 \cdot 4^0 + 2 \cdot 4^1$ and its corresponding encryption is $C = ([1], [2])$. Red arrows indicate bootstrapping.

3.3.3 Tree-based Method

Almost all of the functional bootstrapping methods from the state of the art ([CJP21, KS22, YXS+21, CLOT21, CBSZ23]) take as input a single ciphertext of a message in a relatively small set. In 2021, Guimarães *et al.* [GBA21] specified the *tree-based* and the *chaining* methods for performing functional bootstrapping over several ciphertexts. A natural idea is then to use these methods over input messages decomposed into a smaller basis B. Thus, the encryption of the initial plaintext value is a vector of encryptions of its decomposition digits in basis B. Figure 1 illustrates the tree-based method for the functional bootstrapping of the identity function. First, we create the test polynomials that will be rotated during the BlindRotate step. In the example, the decomposition basis is $B = 4$, so we need to decompose the LUT of the identity function into four polynomials, each with four distinct coefficients. Each coefficient is actually repeated consecutively $\frac{N}{B}$ times to fill the polynomials. Then, we perform four BlindRotate, one on each cleartext polynomial with the first input c_0, followed by four TLWESampleExtract. We get four ciphertexts that we combine together with PublicFunctionalKeyswitch to create a TRLWE encryption of a new test polynomial. Then, we apply a BlindRotate to this encrypted test polynomial with the second encrypted input c_1, and apply a TLWESampleExtract followed by PublicFunctionalKeyswitch to get the final result. In practice, we implement two different PublicFunctionalKeyswitch. The first allows the packing of many TLWE ciphertexts into one TRLWE ciphertext. Meanwhile, the second switches the keys of a TLWE sample. The first key switch has a non-negligible impact on the computation time of a tree-based functional bootstrapping, as seen in Table 1. So, from now on, *we will refer by* $\mathsf{N_{ks}}$ *to the number of calls to PublicFunctionalKeyswitch for TLWE ciphertexts packing into one TRLWE required to evaluate a function on encrypted data.* For the considered example, the tree-based method requires five BlindRotate ($\mathsf{N_{br}} = 5$) and one PublicFunctionalKeyswitch ($\mathsf{N_{ks}} = 1$). For more details about the tree-based functional bootstrapping, the reader is referred to [GBA21].

3.3.4 Multi-Value Bootstrapping

Multi-Value Bootstrapping (MVB) [CIM18] refers to a method for evaluating k different LUTs on a single input at the cost of a single bootstrapping. MVB factors the test polynomial P_{f_i} associated with the function f_i into a product of two polynomials $P_{f_i} = v_0 \cdot v_i$,

where v_0 is a common factor to all P_{f_i}. This factorization allows computing multiple LUTs using a unique blind rotation. Indeed, it is enough to initialize the test polynomial *testv* with the value of v_0 during bootstrapping. Then, we run BlindRotate to get a TRLWE encryption of the polynomial *acc*. We multiply *acc* by each v_i corresponding to the LUT of f_i to get acc_i. Finally, we run a TLWESampleExtract for each acc_i, followed by PublicFunctionalKeyswitch to output k TLWE samples. *From now on, we refer to* $\mathsf{N_{pm}}$ *as the number of multiplications between the plaintext polynomial (v_i) and the TRLWE ciphertext (acc).* So, an MVB requires one BlindRotate ($\mathsf{N_{br}}= 1$) and k plaintext/ciphertext multiplications ($\mathsf{N_{pm}}= k$). More details about the MVB factorization are given in Section 3.3.4. As already noted in [GBA21], the MVB can be applied to the first level of a tree evaluation, as several BlindRotate are performed on different polynomials with the same encrypted input. For instance, regarding Figure 1, instead of requiring five BlindRotate and one PublicFunctionalKeyswitch ($\mathsf{N_{br}}= 5$ and $\mathsf{N_{ks}}= 1$), the tree-based evaluation of the identity function with MVB will only cost two BlindRotate, one PublicFunctionalKeyswitch and four plaintext/ciphertext multiplications ($\mathsf{N_{br}}= 2$, $\mathsf{N_{ks}}= 1$ and $\mathsf{N_{pm}}= 4$). For example, for TFHE parameters associated to $\mathbb{Z}_{16}$ as plaintext space (Table 2), a BlindRotate takes 29 ms, a PublicFunctionalKeyswitch runs in 70 ms and a plaintext/ciphertext multiplication requires 0.1 ms.

4 Choosing the Right Toolbox

4.1 On the Choice of the Functional Bootstrapping Method

4.1.1 Univariate functional bootstrapping

Many works tackled the restriction of TFHE bootstrapping to the evaluation of LUTs of negacylic functions (Sect. 3.3). The half-torus method works around the negacyclic restriction by encoding all the plaintext space $\mathcal{M}$ on $[0, \frac{1}{2})$ (i.e., on the positive half of the torus). As no plaintext values are encoded on the negative half of the torus, any LUT can be encoded within the coefficients of the test polynomial. Then, it is evaluated with only one bootstrapping ($\mathsf{N_{br}}= 1$). Other methods, such as TOTA [YXS+21], FDFB [KS22], or ComBo [CBSZ23], specify several solutions to work around the restriction of working only with half of the torus as a plaintext space. They provide different ways for implementing any LUT with the full torus as plaintext space at the cost of making at least two consecutive BlindRotate ($\mathsf{N_{br}}\geq 2$). However, Clet et al. [CBSZ23] compared all of these methods for the same TFHE parameters and levels of security and showed that the half-torus method achieves the best speed-to-error-rate ratio.

4.1.2 Multivariate functional bootstrapping

In 2021, Guimarães et al. [GBA21] proposed the *tree-based* and *chaining* methods to evaluate LUTs over several encrypted inputs with bootstrappings. These methods can be optimized by using the MVB as discussed in Section 3.3.4. Given a message space of size B, the chaining method requires using a plaintext space of size B^2 with a full torus functional bootstrapping technique or $2B^2$ with the half-torus functional bootstrapping. Meanwhile, the tree-based method (Sect. 3.3.3) requires a plaintext space of size $2 \times B$ and is only meant to be used with the half-torus method. As such, for the chaining method, the size of the parameters dramatically increases with B. This parameter growth jeopardizes the other

speed improvements that could come with the chaining method compared to the tree-based method [TCBS23b]. A recent work by Bon et al. [BPR24] proposes a method to evaluate boolean functions with several encrypted inputs with one bootstrapping. However, their method is limited to binary plaintexts encoded on a small ring $\mathbb{Z}_p$ before encryption. In addition, it requires finding a non-trivial encoding set for the function to be evaluated. Their approach further requires a plaintext domain size dependent on the function's truth table size, which makes it challenging to find an encoding, for example, for adding or multiplying two encryptions of k-bit messages, where a carry must be propagated. Just as recently, [BBB$^+$23] proposed a new programmable bootstrapping operator (WoP-PBS), which inputs several ciphertexts and permits the evaluation of any multivariate LUT. This new method enables efficient bootstrapping of ciphertexts with up to 21-bit precision. However, a follow-up study presented in [BBB$^+$] shows that for 8-bit messages, the tree-based method is *at least* as efficient as the new WoP-PBS independently of the chosen decomposition basis B. In this work, we thus use the tree-based method over the half-torus to compute multivariate 8-bit instructions.

4.2 Optimal Basis Selection for LUT Evaluation

4.2.1 Decomposition Basis Choice

The message space corresponding to 8-bit messages is the set $\mathcal{M} = \{0, 1, \cdots, 255\}$. Since we use the half-torus bootstrapping method, we have to work on a 512-element discretized torus. This requires very large TFHE parameters leading to a very slow bootstrapping ($\approx$1.5 secs for a single bootstrapping [TCBS23b]). Consequently, we need to break down our 8-bit data into a smaller basis. For 8-bit plaintexts, several decompositions are available: we can decompose a message into four 2-bit digits, into three 3-bit digits (with the most significant one only taking values in $\{0, \cdots, 3\}$), or into two 4-bit digits. For instance, basis 16 allows the decomposition of 8-bit messages into two nibbles. Note that the smaller the decomposition basis, the smaller the parameters, and thus the faster the bootstrapping evaluation. However, the smaller the decomposition basis, the greater the number of digits, and so the greater is the number of bootstrapping to be performed. A tradeoff must, therefore, be achieved between the number of bootstrapping needed and the parameters' size corresponding to the decomposition basis. *We refer to the evaluation of the tree-based method (using MVB) on an 8-bit message decomposed into d digits in basis B as* LUTeval, as opposed to SimpleBoot, which is the usual bootstrapping operation taking only one encrypted input. The bootstrapping cost of LUTeval is $NB_{boot} = 1 + \sum_{i=0}^{d-2} B^i$, where the 1 comes from the trick of computing the output of the first level of the tree with MVB instead of running B^{d-1} bootstrappings (Sect. 3.3.4). To obtain the d digits forming the result of evaluating a LUT from $\mathcal{M}$ to $\mathcal{M}$, LUTeval must be performed d times on the same inputs. That is why we further introduce MVLUTeval, which uses the MVB optimization to reduce the number of BlindRotate $\mathsf{N_{br}}$. As seen in Table 1, when run under TFHElib [CGGI16] with the parameters from Table 2, the SimpleBoot is the most efficient for basis 4. However, in the sequel, the most used operators are LUTeval and different flavors of MVLUTeval. The best timings for these operations are obtained with decomposition basis 16. As a matter of illustration, evaluating two LUTeval in basis 16 costs 0.26 seconds. So MVLUTeval*, which does the same thing with one less BlindRotate, takes 0.23 seconds. However, MVLUTeval is less interesting in other bases, where the initial number of BlindRotate is larger: MVLUTeval* saves one BlindRotate, i.e., $\frac{1}{4}$ in basis 16, but only $\frac{1}{20}$ in basis 8 and $\frac{1}{44}$ in basis 4. Note that for binary operators, basis 16 is not optimal. Indeed, these operations can be implemented

with depth-2 tree-based bootstrapping regardless of the decomposition basis. For bases 2, 4, and 8, this respectively leads to 8, 8, and 6 blind rotations vs 4 for basis 16. On the other hand, for any operation requiring calls to LUTeval, basis 16 remains the most efficient. For example, for the addition, which is the most straightforward bivariate operation apart from bitwise ones, the number of bootstrappings required to propagate the carry with decomposition basis 4 is such that the evaluation of the addition takes just as long as for basis 16. For all other non-bitwise functions, basis 16 is the most efficient. So, despite the better efficiency of basis 4 for bitwise operators, basis 16 is the optimal choice. †

Table 1: Execution times of SimpleBoot, LUTeval and MVLUTeval depending on the plaintext decomposition basis. $\mathsf{MVLUTeval}^{\star}$ stands for an evaluation of two different LUTs, and $\mathsf{MVLUTeval}^{\diamond}$ for four different LUTs. As the number of output digits is equal to the number of LUT evaluations, $\mathsf{MVLUTeval}^{\star}$ outputs two new basis B ciphertexts and $\mathsf{MVLUTeval}^{\diamond}$ outputs four such ciphertexts. For instance, for basis 16, $\mathsf{MVLUTeval}^{\star}$ (respectively $\mathsf{MVLUTeval}^{\diamond}$) thus output 8 bits (respectively 16 bits), or equivalently a "digit" in basis 256 (respectively 256×256).

Decomposition basis		LUT size	Number of output digits	Corresponding output basis	$\mathsf{N_{br}}$	$\mathsf{N_{ks}}$	Timings (secs)
16	SimpleBoot	16	1	16	1	0	0.029
	LUTeval	256	1	16	2	1	**0.13**
	$\mathsf{MVLUTeval}^{\star}$	256	2	256	3	2	**0.23**
	$\mathsf{MVLUTeval}^{\diamond}$	256	4	256×256	5	4	**0.43**
8	SimpleBoot	8	1	8	1	0	0.015
	LUTeval	256	1	8	10	9	0.47
	$\mathsf{MVLUTeval}^{\star}$	256	2	64	19	18	0.93
	$\mathsf{MVLUTeval}^{\diamond}$	256	4	256×8	37	36	1.83
4	SimpleBoot	4	1	4	1	0	**0.007**
	LUTeval	256	1	4	22	21	0.5
	$\mathsf{MVLUTeval}^{\star}$	256	2	16	43	42	0.993
	$\mathsf{MVLUTeval}^{\diamond}$	256	4	256	85	84	1.98

4.2.2 LUT Dereferencing Operators

Now that we have settled on the optimal decomposition basis for our 8-bit plaintext inputs, we can instantiate our LUT dereferencing tools SimpleBoot, LUTeval, and MVLUTeval. The first is the basic TFHE bootstrapping with a 4-bit ciphertext as an encrypted index. Let tab_16 be a cleartext LUT with 16 entries in $\mathbb{Z}_{16}$, given a ciphertext $c \in [\![m]\!]$, $\mathsf{SimpleBoot}(c;$ tab_16) returns $c' \in [\![\mathsf{tab\_16}[m]]\!]$. The second allows us to evaluate a 16×16 LUT on two ciphertexts $c_0 \in [\![m_0]\!]$ and $c_1 \in [\![m_1]\!]$, with $m_0, m_1 \in \mathcal{M} = \{0, 1, \cdots, 15\}$. We note it $\mathsf{LUTeval}(c_0, c_1;$ tab), with tab the 16×16 table that will be used to instantiate the 16 test-vectors polynomials required for the tree-based bootstrapping. $\mathsf{LUTeval}(c_0, c_1;$ tab) returns a 4-bit ciphertext $c' \in [\![\mathsf{tab}[16m_0 + m_1]]\!]$. Lastly, let us assume that we want to evaluate k LUTeval on the following pairs of ciphertexts $((c_\alpha, c_1), \cdots, (c_\alpha, c_k))$ using the tables (tab_1,$\cdots$,tab_k). Each pair (c_α, c_j) is an encryption of $T_j = 16m_\alpha + m_j$, where $m_\alpha, m_j \in \mathbb{Z}_{16}$. As c_α is a common input for the k LUTeval, we can rely on only one MVB to compute

the first level of the k trees simultaneously instead of running k separate MVB for each $\mathsf{LUTeval}(c_\alpha, c_j; \mathsf{tab\_j})$, where $j \in \{1, \cdots, k\}$. The second level of each tree is then computed separately on $(c_1, \cdots, c_k)$. As such, we end up running $k+1$ $\mathsf{BlindRotate}$ ($\mathsf{N_{br}}= k+1$) instead of $2k$ ones for computing k $\mathsf{LUTeval}$, with k $\mathsf{PublicFunctionalKeyswitch}$ ($\mathsf{N_{ks}}= k$) and $16k$ plaintext/ciphertext multiplications ($\mathsf{N_{pm}}= 16k$).

From now on, we define $\mathsf{MVLUTeval}(c_\alpha;\, c_1, \cdots, c_k;\, \mathsf{tab\_1}, \cdots, \mathsf{tab\_k})$ as the operation that computes with a unique MVB the first level of the trees associated to $\mathsf{LUTeval}(c_\alpha, c_j;\, \mathsf{tab\_j})$, and outputs k encrypted 4-bit digits $c'_j \in [\![\mathsf{tab\_j}\,[16m_\alpha + m_j]]\!]\ \forall j \in \{1, \cdots, k\}$. $\mathsf{MVLUTeval}$ can be further optimized when provided with the same table $\mathsf{tab\_j}$ twice (or more) by computing less $\mathsf{PublicFunctionalKeyswitch}$.

Table 2: Parameters set for the considered decomposition basis ($\lambda \approx 128$). B_g and l denote the basis and levels associated with the gadget decomposition, B_{KS} and t denote the decomposition basis and the precision of the decomposition of the $\mathsf{PublicFunctionalKeyswitch}$, r denotes the plaintext modulus, and ϵ is the error probability of one MVB tree-based evaluation. The unitary TFHE ciphertext size is given by $n \log_2(q)$, leading for example to an overall ciphertext size of 65600 bits to represent 2 basis-16 digits.

B	n	q	N	l	B_g	B_{KS}	t	r	ϵ	TRLWE std	TLWE std
4	700	2^{32}	1024	5	16	1024	2	8	2^{-30}	5.6×10^{-8}	1.9×10^{-5}
8	700	2^{32}	2048	2	2048	1024	2	16	2^{-23}	9.6×10^{-11}	1.9×10^{-5}
16	1024	2^{32}	2048	3	256	1024	2	32	2^{-23}	9.6×10^{-11}	6.5×10^{-8}

In summary, our toolbox mainly consists of $\mathsf{LUTeval} : \mathcal{C}^2 \times \mathcal{L} \longrightarrow \mathcal{C}$ ($\mathcal{L}$ being the set of all 256 4-bit entries tables) which, given $(c_0, c_1) \in [\![m_0]\!] \times [\![m_1]\!]$, is such that

$$\mathsf{LUTeval}(c_0, c_1; \mathsf{tab}) \in [\![\mathsf{tab}[16m_0 + m_1]]\!].$$

and $\mathsf{MVLUTeval} : \mathcal{C}^{(k+1)} \times \mathcal{L}^k \longrightarrow \mathcal{C}^k$, its optimization for running several $\mathsf{LUTeval}$ with one common input $c_\alpha \in [\![m_\alpha]\!]$ and k other inputs $c_j \in [\![m_j]\!], \forall j \in \{1, \cdots, k\}$, which satisfies

$$\mathsf{MVLUTeval}(c_\alpha; c_1, \cdots, c_k; \mathsf{tab}_1, \cdots, \mathsf{tab}_k) \in [\![\mathsf{tab}_1[16m_\alpha + m_1]]\!] \times \cdots \times [\![\mathsf{tab}_k[16m_\alpha + m_k]]\!].$$

5 An FHE-Optimized Instruction Set

5.1 Instruction Set Overview

In this paper, we propose an exhaustive set of some fifty 8-bit instructions that manipulate (T)FHE-encrypted data. Some provided instructions are relatively standard, but others are more specific and included because a smaller number of homomorphic operations are required to implement them. As an example of this, for additions, we provide three instructions: ADD, ADDi, and ADDZ. The ADD instruction takes *two* input ciphertexts (with an 8-bit cleartext payload) and, without surprise, produces a third one whose decryption is expected to be the sum of the two input ciphertexts' plaintexts. The ADDi instruction takes *one* input ciphertext and an immediate (public) value V. This instruction can then be seen as a family of *univariate* instructions ADDi_V (for $V = 0, \cdots, 255$) and, as we shall later see, can be much faster implemented than the previous general purpose ADD. Lastly, the ADDZ instruction also takes *two* input ciphertexts and performs an addition *under the assumption that at least one of the two input ciphertexts is an encryption of 0.* This case

occurs recurrently in several algorithmic patterns, particularly when values must be selected based on the results of conditions over encrypted data. Some examples are the computation of conditional assignment instruction CSEL (Section 4.4), bubble sorting (Section 8, p. 22), array dereferencing and assignment (Section 8, p. 23). For more details about ADDZ, see Section B.2.2. As a result, this instruction also executes much faster than the general purpose ADD instruction. This first example illustrates our design mindset, according to which we have proposed standard general-purpose instructions for all usual operations found in typical ISA, as well as additional variants providing better FHE evaluation when some (frequently occurring) assumptions are met.

In summary, we provide the following categories of instructions:

- Bitwise/arithmetic instructions (addition, multiplication, division, modulo, shift, rotation, etc.), each coming in different flavors as discussed just above. These instructions' names are relatively conventional.
- Test instructions for testing equality and performing comparisons over encrypted data. These instructions also come with different flavors and are expected to return encryptions of either 0 or 1 .
- Conditional assignment instructions (CDUP, NCDUP and CSEL, the latter being the only trivariate instruction in the set). These instructions provide the building blocks to emulate if-then-else or do-while statements with encrypted data-dependent conditions.
- Advanced instructions: support for multiplication with 16-bit results (i.e., computation of the most significant *byte* of the product of two bytes), support for fixed-point arithmetic (including decimal division), min/max operators, absolute value, to name a few.
- User defined *univariate* instructions: we further provide an XOP instruction which the programmer may arbitrarily configure.

For readability's sake, the following sections are intended only to discuss the key difficulties we had to overcome and the optimization techniques we had to consider to implement the complete set of instructions. Full details are provided in Appendix B.

5.2 Notations for Homomorphic Operator Specifications

In this work, following Section 4 and most particularly Sect. 4.2, we manipulate 8-bit plaintexts broken down into two 4-bit digits. Thus, to encrypt an 8-bit plaintext M decomposed into two 4-bit digits m_0 and m_1 such that $M = 16m_0 + m_1$, we encrypt m_0 and m_1 separately under the same scheme $\mathcal{E}$ to obtain $C = (c_0, c_1) \in [\![m_0]\!] \times [\![m_1]\!]$ as an encryption of M. We consistently denote 8-bit plaintexts $M \in \mathcal{M}^2$ and their corresponding ciphertexts $C \in \mathcal{C}^2$ with uppercase letters. Conversely, 4-bit plaintexts and their encryptions are denoted with lowercase letters. For instance, for $h, l \in \mathcal{M}^2$, $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!] \subset \mathcal{C}^2$ denotes an encryption of the 8-bit cleartext value $(h, l) \in \mathcal{M}^2$ which encodes the 8-bit message $M = 16h + l$. We call h the *most significant nibble* of M. Similarly, l is the *least significant nibble* of M. We denote these parts MSN and LSN. With a slight abuse of notation, as already done above, we will use $T = (u, v)$ and $T = 16u + v$ interchangeably. Sometimes, a ciphertext C may have no MSN and is denoted by $(\perp, c_1)$. This, for example, occurs for outputs of test instructions, which are encrypted booleans (in that case, it can further be assumed that $c_1 \in [\![0]\!] \cup [\![1]\!]$). Some instructions also result in a cleartext 0 value in the MSN or LSN of a given ciphertext, e.g., an unsigned right (respectively left) shift of $C = (c_0, c_1)$ gives ciphertext $(0, c_0)$ (respectively $(c_1, 0)$). We can use this to perform cleartext/ciphertext operations on the fly.

As a "Hello world!" example of how we later use these notations to specify our operators and instructions, let us consider the AND instruction which, given $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and $C = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$, is defined as

$$\mathsf{Eval}(\mathsf{AND}; C, C') = \bar{C} = (\bar{c}_0, \bar{c}_1) \in [\![h \& h']\!] \times [\![l \& l']\!]$$

To actually *implement* the above, we then proceed by evaluating

$$\bar{c}_0 = \mathsf{LUTeval}(c_0, c'_0; \mathsf{tab\_and}) \text{ and } \bar{c}_1 = \mathsf{LUTeval}(c_1, c'_1; \mathsf{tab\_and})$$

where tab_and is a table with 256 4-bit entries such that $\mathsf{tab\_and}[16i + j] = i \& j$ and where $\mathsf{LUTeval} : \mathcal{L} \times \mathcal{C}^2 \longrightarrow \mathcal{C}$ ($\mathcal{L}$ being the set of all 256 4-bit entries tables) is the tree-based functional bootstrapping operator instantiated in Section 4.2.

5.3 Implementing Univariate Instructions

Univariate instructions only take *one* input ciphertext (with an 8-bit cleartext payload). These can correspond to univariate operators, such as the absolute value (ABS) or the negation (NEG) of a signed 8-bit value, the bitwise inversion operator (INV), etc. They can also correspond to cleartext-ciphertext operations such as the addition of a (public) immediate value (ADDi), left shift, or rotation by a (public) number of positions (SHLi or ROLi), etc. *With respect to our 8-bit plaintext domain*, all these operations can be implemented by simply dereferencing a table with 256 8-bits entries with an 8 bits plaintext input, i.e., any such instruction inst on input $i \in \mathbb{Z}_{256}$ can be implemented as $\mathsf{tab\_inst}[i]$ with $\mathsf{tab\_inst}[i] = f(i)$ (for $i = 0, \cdots, 255$) and f the function that inst performs. For instructions implementing cleartext-ciphertext operations, there is one such table $\mathsf{tab\_inst}_V$ for each of the 256 possible plaintext inputs, V, with the proper table selected at runtime (and, even possibly generated on the fly). Then, to perform the instruction inst over $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ we simply have to evaluate

$$(\bar{c}_0, \bar{c}_1) = \mathsf{MVLUTeval}(c_0; c_1, c_1; \mathsf{tab\_inst\_msn}, \mathsf{tab\_inst\_lsn}) \tag{1}$$

with $\mathsf{tab\_inst\_msn}[i] = \lfloor \mathsf{tab\_inst}[i]/16 \rfloor$ and $\mathsf{tab\_inst\_lsn}[i] = \mathsf{tab\_inst}[i] \pmod{16}$, for $i = 0$ to 255. To illustrate that this pattern allows implementing arbitrary complex univariate instructions, we can consider the case of the divide-by-V ($V \in \mathbb{Z}_{256}$) operation[9] which induces the instructions: DIVi (quotient of the euclidean division by V with $\mathsf{tab\_divi}_V[i] = \lfloor i/V \rfloor$), MODi (remainder of the euclidean division by V with $\mathsf{tab\_modi}_V[i] = i \mod V$).

Univariate test instructions are handled slightly differently in the sense that, with respect to the plain domain, they output only encryptions of boolean 1-bit values (still contained in a single 4-bit digit). As such, only an evaluation of LUTeval is needed to perform them. For example, the $\mathsf{LT}(C, V)$ instruction, which outputs ciphertext $\bar{C} = (\perp, c_1) \in \{\perp\} \times [\![b]\!]$ from ciphertext $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ with $b = 1$ if $16h + l < V$ and $b = 0$ otherwise, is performed by evaluating only

$$\bar{c}_1 = \mathsf{LUTeval}(c_0, c_1; \mathsf{tab\_lt}_V). \tag{2}$$

[9] Division, even by a cleartext value, is a good example of an operation which is notoriously difficult to perform efficiently over FHE (even when one of the two operands is cleartext). Here, with our techniques, division by a cleartext value does not cost much more than a mere addition...

Note that some univariate instructions can be implemented more efficiently than by (1). For example, for an addition by V (ADDi) we can proceed as follow:

$$\bar{c}_1 = \mathsf{SimpleBoot}(c_1, \mathsf{tab\_add4}_{V\&15})$$
$$\bar{c}_0 = \mathsf{LUTeval}(\bar{c}_0, c_1, \mathsf{tab\_fin4})$$

with $\mathsf{tab\_add4}_v[i] = (i+v) \mod 16$, $\mathsf{tab\_fin4}_V[16i+j] = \mathsf{tab\_add4}[16 \times \mathsf{tab\_add4}_{\lfloor V/16 \rfloor}[i] + \mathsf{tab\_car4}_{V\&15}[j]]$, $\mathsf{tab\_car4}_v[i] = \lfloor (i+v)/16 \rfloor$ and $\mathsf{tab\_add4}[16i+j] = (i+j) \mod 16$. Following the notations in Sect. 5.2, this can further be optimized as

$$(\bar{c}_0, \bar{c}_1) = \mathsf{MVLUTeval}(c_1; \perp, c_0; \mathsf{tab\_add4}_{V\&15}, \mathsf{tab\_fin4})$$

which has the effect of factoring an additional blind rotation (resulting in 2 blind rotations vs 3 if (1) is used). In a similar spirit, bitwise instructions, e.g. ANDi, can simply be implemented with two calls to SimpleBoot leading, again, to 2 blind rotations vs 3 when (1) is used. We have considered such optimizations on a case-by-case basis, resorting to (1) only when we found no better options. †

Lastly, we provide an additional univariate XOP instruction taking a user-defined 256×8 bits table rather than an immediate value V as input. For example, this instruction can perform special operations such as the AES S-box or the six $GF(256)$ multiplication-by-cleartext in that algorithm [TCBS23b]. A variant of this latter instruction, XOPN(ibble) also takes a user-defined 256×4 bits table as input to evaluate custom conditions following (2). Table 3 provides a synthetic (yet exhaustive) list of the univariate instructions we have implemented.

Arithmetic inst.	ADD(i) (addition of two bytes); SUB(i); MUL(i); MULM(i) (most sig. *byte* of the product of two bytes); DIV4(i) (division of an encrypted byte by an encrypted nibble); DIV(i) (division of an encrypted byte by another encrypted one); MOD4(i) (modulo of an encrypted byte by an encrypted nibble); MOD(i) (modulo of an encrypted byte by another encrypted byte)
Bitwise inst.	AND(i); OR(i); (U)SHL(i) (shift an encrypted byte (signed or unsigned) left by an encrypted 8-bit index), ROL(i) (rotate an encrypted byte left by an encrypted 8-bit index); (U)SHR(i); ROR(i)
Test inst.	EQ(i) (test if two ciphertexts encrypt the same byte); GT(i) (test if the first ciphertext encrypts an 8-bit value greater than the one encrypted by the second ciphertext); LT(i); GTE(i); LTE(i)
Other inst.	MIN(i) (minimum of two encrypted bytes); MAX(i); CDUP(i); NCDUP(i); CSEL (conditional selection); ABS (absolute value of an encrypted signed byte); NEG (returns the opposite of an encrypted signed byte); XOP; XOPN; DC (binary decomposition of a ciphertext); RC (recomposition of a ciphertext)

Table 3: List of our instructions. For each instruction denoted by INSTR(i), INSTR is the bivariate instruction taking two encrypted inputs, and INSTRi is the variant taking as inputs an encryption of a byte and a cleartext one. Instructions denoted by (U)INSTR(i) have an unsigned and a signed version.

5.4 Implementing Bivariate Instructions

5.4.1 Bivariate Instructions Basics

We now turn to bivariate instructions, which take *two* input ciphertexts (each with an 8-bit cleartext payload). These correspond to additions (ADD), left shift or rotation by an encrypted number of positions (SHL or ROL), etc. In Section 5.2, we have already seen how to perform bitwise instructions. As another example, let us consider instruction ADD which turns $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$ into $\bar{C} = (\bar{c}_0, \bar{c}_1) \in [\![\bar{h}]\!] \times [\![\bar{l}]\!]$ such that $16\bar{h} + \bar{l} = (16h + l + 16h' + l') \mod 256$ leading to two calls to LUTeval and one call to MVLUTeval with two tables to produce $\bar{C}$:

$$\begin{aligned} c_s &= \mathsf{LUTeval}(c_0, c'_0, \mathsf{tab\_add}) \\ (\bar{c}_1, c_c) &= \mathsf{MVLUTeval}(c_1; c'_1, c'_1; \mathsf{tab\_add}, \mathsf{add\_carry}) \\ \bar{c}_0 &= \mathsf{LUTeval}(c_s, c_c, \mathsf{tab\_add}). \end{aligned} \tag{3}$$

with $\mathsf{tab\_add}[16i + j] = (i + j) \mod 16$ and $\mathsf{add\_carry}[16i + j] = \lfloor \frac{i+j}{16} \rfloor$.

As for the univariate case (Section 5.3), bivariate test instructions output only a single (encrypted) nibble with a single-bit payload, i.e., ciphertexts of the form $(\perp, c_1)$ with $c_1 \in [\![0]\!] \cup [\![1]\!]$. In FHE computations, we often have to sum two encrypted values where an unknown one of them encrypts 0. This does not make any difference on a cleartext processor, and special instructions are usually not included in that case. However, when working over encrypted data, the lack of carry propagation means we can save the call to MVLUTeval in Eq. (3) above. For this reason, we also provide the ADDZ instruction, which thus "sums" two ciphertexts under the assumption that at least one of them belongs to $[\![0]\!]$ by means of two independent calls to LUTeval.

As we shall see, the CDUP ("Conditional DUPplication") instruction plays an important role in being able to perform a conditional assignment and, as such, is fundamental in the context of FHE calculations. Given an encrypted boolean $(\perp, c_1) \in [\![l]\!]$ and an input $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$, CDUP produces ciphertext $\bar{C} = (\bar{c}_0, \bar{c}_1) \in [\![\bar{h}]\!] \times [\![\bar{l}]\!]$ with $\bar{h} = h'$ and $\bar{l} = l'$ when $l = 1$ (i.e. when the input boolean is true) and $\bar{h} = \bar{l} = 0$ when $l = 0$ (we leave the instruction behavior unspecified when $l > 1$). Essentially, CDUP can be implemented by a single call to MVLUTeval:

$$\mathsf{CDUP}((\perp, c_1), C') = \mathsf{MVLUTeval}(c_1; c'_0, c'_1; \mathsf{tab\_sel1}, \mathsf{tab\_sel1}),$$

with $\mathsf{tab\_sel1}[16i + j] = j$ if $i = 1$ and 0 otherwise. Conversely, instruction NCDUP behaves similarly except that it outputs (encryption of) 0 when the input boolean is true. As such, it is implemented exactly as CDUP but using table $\mathsf{tab\_sel0}[16i + j] = j$ if $i = 0$ and 0 otherwise. Lastly, we provide a single *trivariate* instruction CSEL ("Conditionnal SELection") which, given an encrypted boolean $(\perp, c_1) \in [\![b]\!]$ ($b \in \{0, 1\}$) and *two* inputs $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$ and $C'' = (c''_0, c''_1) \in [\![h'']\!] \times [\![l'']\!]$, produces ciphertext $\bar{C} = (\bar{c}_0, \bar{c}_1) \in [\![\bar{h}]\!] \times [\![\bar{l}]\!]$ such that $\bar{h} = bh' + (1 - b)h''$ and $\bar{l} = bl' + (1 - b)l''$. Interestingly, even if it is a trivariate instruction, CSEL can be implemented rather efficiently by factoring 4 blindRotate in a single call to MVLUTeval:

CSEL

$$\begin{aligned} (\tilde{c}_0, \tilde{c}_1, \tilde{c}_2, \tilde{c}_3) &= \mathsf{MVLUTeval}(c_1; c'_0, c'_1, c''_0, c''_1; \mathsf{tab\_sel1}, \mathsf{tab\_sel1}, \mathsf{tab\_sel0}, \mathsf{tab\_sel0}) \\ (\bar{c}_0, \bar{c}_1) &= \mathsf{ADDZ}((\tilde{c}_0, \tilde{c}_1), (\tilde{c}_2, \tilde{c}_3)) \end{aligned}$$

This gives us the conditional assignment instruction needed to emulate if-then-else constructs on our FHE processor abstraction. Note that we also provide instructions CDUPi, NCDUPi, and CSELi, which all take an encrypted boolean as input and either one or two cleartext values. We detail these instructions in Appendix B.

5.4.2 A Homomorphic Division Operator

We now illustrate how our approach can be used to lead to a division operator between two ciphertexts $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$. For simplicity's sake, we consider the unsigned division. This operator returns $\bar{C} = (\bar{c_0}, \bar{c_1}) \in [\![\bar{h}]\!] \times [\![\bar{l}]\!]$ such that $16\bar{h} + \bar{l} = \lfloor (16h + l)/(16h' + l') \rfloor$.

Let $16\bar{h} + \bar{l} = Q = \sum_{i=0}^{7} q_i 2^i = \sum_{i=0}^{7} p_i$ with $q_i \in \{0, 1\}$. We then have

$$q_i = \begin{cases} 1 & \text{if } 2^i(16h' + l') \leq (16h + l) - \sum_{j=i+1}^{7} q_j \times (16h' + l')2^j, \\ 0 & \text{otherwise.} \end{cases}$$

Then, Q can be naively obtained using a carryless summation of the $p_i = q_i 2^i$'s. But we note that if $h' \neq 0$, the MSN of $\frac{16h+l}{16h'+l'}$ is always 0. If $h' = 0$, then the MSN of the quotient is given by $\frac{h}{l'}$. So instead of computing the q_i's for $i \in \{7, 6, 5, 4\}$, it is sufficient to compute $\bar{c_0}$ = LUTeval(SimpleBoot(c'_0,tab_is_zero), LUTeval(c_0, c'_1;tab_div);tab_mul). Then, to compute the LSN of the quotient, we must follow the algorithm presented below, starting from C_t rather than C. Indeed, after computing $\bar{c_0}$, the value that will be compared to obtain the LSN of the quotient must be updated in the following way. First, we need to compute c_s = LUTeval($\bar{c_0}, c'_1$;tab_mul), and then update c_0 with the value c_t = LUTeval(c_0, c_s,tab_sub).

DIV

// the ciphertext $C_t = (c_t, c_1)$ has been previously computed following the method presented above
$C_t = (c_t, c_1)$
$c_q \in [\![0]\!]$
for $i = 3$ to 0
  $C_m = (c_{m_0}, c_{m_1})$ = SHLi(C', i) // Shift Left by a cleartext index
  c_g = GTE(C_t, C_m) // Greater Than or Equal
  c_b = LUTeval(c'_0, c_g, tab_and_mulm_zero)
  C_s = MVLUTeval($c_b; c_{m_0}, c_{m_1}$; tab_mul_lsn, tab_mul_lsn)
  C_t = SUB(C_t, C_s)
  c_q = LUTeval(c_q, c_b, tab_add_q_i)

---

tab_and_mulm_zero is a 256-element table with tab_and_mulm_zero$[16k + j]$ = $(((k << i) >> 4) == 0) \& (j == 1)$, that we use to test if the overflow produced by the multiplication $2^i(16h' + l')$ is zero and if $2^i(16h' + l') \leq (16h + l) - \sum_{j=i+1}^{7} q_j 2^j$. Indeed, the condition for $q_i = 1$ is satisfied if and only if the multiplication does not produce any overflow. tab_add_q_i is a 256-element table such that for $k, j \in \{0, \cdots, 15\}$, tab_add_$q_i[16k + j] = k + (j \neq 0) \cdot 2^i$ that we use to add the new q_i to c_q. Finally, note that for $i = 0$, SHLi does nothing.

Let us further consider the case where it is known that $h' = 0$ and let $q_0 = \lfloor 16h/l' \rfloor$, $q_1 = \lfloor l/l' \rfloor$, $r_0 = 16h \mod l'$ and $r_1 = l \mod l'$, then the division algorithm may be significantly

simplified due to the following relation, which holds $\forall (h, l, l') \in \{0, \cdots, 15\}^3$,

$$\left\lfloor \frac{16h + l}{l'} \right\rfloor = \underbrace{\left\lfloor \frac{16h}{l'} \right\rfloor}_{q_0} + \underbrace{\left\lfloor \frac{l}{l'} \right\rfloor}_{q_1} + \underbrace{\left\lfloor \frac{16h_r}{l'} \right\rfloor}_{\epsilon_0} + \underbrace{\left\lfloor \frac{l_r}{l'} \right\rfloor}_{\epsilon_1}, \tag{4}$$

with $h_r = \left\lfloor \frac{r_0 + r_1}{16} \right\rfloor$ and $l_r = (r_0 + r_1) \mod 16$. This simplified division requires 20 blind rotations and 12 key switches versus 97 and 56 for a full-blown division. See Table 4. †

6 Other Types of Ciphertexts

6.1 Working with Signed Inputs

In this section, we consider 8-bit messages decomposed into two 4-bit digits, but in signed representation using two's complement. This way, we can encrypt messages in $\mathcal{M}^- = [\![-128, 127]\!]$. This, of course, requires the user to know whether they are using signed or unsigned representation to be able to interpret the decrypted messages correctly. Note that the used TFHE parameters do not change, as it is only a matter of semantics. This way, new tables are required to perform additions, multiplications, shifts,... and new operations such as the negation NEG or the absolute value ABS. For many operators, these operations are very similar to their unsigned variants. For instance, the signed right logical shift (SHRi) is implemented by duplicating the sign bit instead of injecting zeroes on the left. Thus, the only differences are in the tables tab_inst_msn and tab_inst_lsn (recall Equation (1)) contents. Apart from this, we proceed similarly to the unsigned shift. For further details on how we implement SHRi and SHR, see Section B.2.8. More generally, appendix B presents all our instructions implementation details, including those instructions working over signed data.

6.2 Support for Fixed-point Arithmetic

We can also apply this paper's approach to ciphertexts encrypting values represented in fixed-point arithmetic. To do so, we have to work with 16-bit data: 8 bits for the integer part and 8 bits for the fractional part of a fixed point number. In addition, we need an encoding layer adapted to the semantic of this representation on top of the encryption layer. We consider that the integer part can be signed or unsigned and that the fractional part is always positive. For example, 4.6 is represented as $(4, \lfloor 256 \times 0.6 \rfloor = 153)$ and -4.6 as $(-5, \lfloor 256 \times 0.4 \rfloor = 102)$. We note the ciphertexts and associated plaintexts corresponding to encryptions of such 16-bit messages with bold capital letters: $\boldsymbol{C} = (c_0, c_1, c_2, c_3) \in [\![h]\!] \times [\![l]\!] \times [\![o]\!] \times [\![k]\!]$ is an encryption of the 16-bit message $\boldsymbol{T}$ encoding $16h + l + \frac{16o+k}{256}$. For example, an encryption of $\frac{1}{256} = 0.00390625$ will be $\boldsymbol{C} \in [\![0]\!] \times [\![0]\!] \times [\![0]\!] \times [\![1]\!]$. This approach enables the implementation of new functions, such as decimal division or a fixed-precision sigmoid (Section 8). As an example, let us consider the decimal division by a cleartext 8-bit value d (assuming unsigned input semantic for simplicity's sake), an operation that is often used when computing basic statistics when the sample size is known, which given $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ outputs $\bar{C} = (\bar{c}_0, \bar{c}_1) \in [\![\bar{h}]\!] \times [\![\bar{l}]\!]$ and $(\tilde{c}_0, \tilde{c}_1) \in [\![\tilde{h}]\!] \times [\![\tilde{l}]\!]$ such that

$$16\bar{h} + \bar{l} = \mathsf{tab\_int}[16h + l] = \left\lfloor \frac{16h + l}{d} \right\rfloor \text{ and } 16\tilde{h} + \tilde{l} = \mathsf{tab\_dec}[16h + l] = \left\lfloor 256 \frac{(16h + l) \mod d}{d} \right\rfloor.$$

With only 5 BlindRotate, we then compute $(\bar{c}_0, \bar{c}_1, \tilde{c}_0, \tilde{c}_1)$ as the result of

$$\mathsf{MVLUTeval}(c_0; c_1, c_1, c_1, c_1; \mathsf{tab\_int\_msn}, \mathsf{tab\_int\_lsn}, \mathsf{tab\_dec\_msn}, \mathsf{tab\_int\_lsn})$$

6.3 Input/Output

In this section, we consider the issue of getting data in and out of our processor abstraction in the setting where it is deployed on a remote server and available to a *client*, which sends input ciphertexts to the server (which we refer to as *uplink* input transmissions from the client to the server) and expects output ciphertexts in return (which we refer to as *downlink* output transmissions from the server back to the client), as the results of some valuable computations. We then wish to avoid the naive approach, which consists of the client sending its encrypted input data by transferring a full TLWE ciphertext for each payload nibble (similarly on the downlink).

Still, in this naive setting (and considering the parameters in Table 2), our approach is more efficient than the "standard TFHE gate approach" as we require transmitting two TFHE ciphertexts with a 4-bit payload (for a total size of 65600 bits) by opposition to eight ciphertexts with a single bit payload (accounting for a total of 161536 bits using the default TFHELib parameters).

6.3.1 Uplink Input Data Transmission

On the uplink, a standard approach is to resort to transciphering to remove the transmission overhead [CCF+16, BBS22, BCBS23, PJH23], at the cost of homomorphically running a symmetric algorithm decryption function (which can then be easily "coded" using our instruction set). If we accept a slightly higher transmission overhead, a computationally lighter approach consists of simply synchronizing the client and server using a PRF to avoid sending the a term of the TLWE pairs (i.e., both the server and the client are able to compute on their own the a vector associated to a given $b = \langle a, s\rangle + \frac{q}{16}m + e$) and thus to transmit only the unique coefficient b. The uplink expansion factor, therefore, becomes independent of the n parameter. Since the ciphertext and plaintext moduli, respectively, are $q = 2^{32}$ and $B = 16$ in our TFHE parameter setting for basis-16 (Table 2), this leads to an expansion factor of only $\frac{32}{4} = 8$, which is reasonable by "FHE standards".

6.3.2 Downlink Output Data Transmission

Remark that none of the above approaches are applicable to reduce the overhead of encrypted results transmission from the server to the client. Indeed, transciphering allows the conversion of data encrypted under some (usually symmetric) scheme towards an *homomorphic* scheme, but not the other way around. Besides, for results of FHE computations, neither the server nor the client can control the resulting a term, which, therefore, has to be transmitted somehow. Still, to decrease as much as possible the burden of transmitting several encrypted outputs, under the form of TLWE ciphertexts, the server can assemble them as much as possible in TRLWE ones. Thus, we want to assemble $K = 2L$ TLWE results into one or more TRLWE ciphertexts. More precisely, the server assembles up to n TLWE samples into a single TRLWE sample using the usual keyswitch packing whereby n TLWE messages $m_0, \cdots, m_{n-1}$ maps to $m(X) = \sum_0^{n-1} m_i X^i$. Consider that K TLWE ciphertexts

have to be transmitted, then, when $K \mod n = 0$, we have an expansion factor of

$$\frac{2\log_2 q}{\log_2 t} \tag{5}$$

i.e., with $n = 1024$ and $q = 2^{32}$, this leads to an expansion factor of 16 ($B = 16$). So, the downlink expansion factor is "only" twice that of the uplink (asymptotically). When, $K \mod n = r > 0$, expansion is given by

$$\frac{2\lfloor k/n \rfloor \log_2 q + (n+r)\log_2 q}{K \log_2 t}.$$

Expansion factor (5) is also valid in the asymptotic regime when K is large. Other techniques may be used to further reduce the expansion factor on the downlink, e.g., [BDGM19, BCCS24].

6.4 Bit decomposition and recomposition

6.4.1 Decomposition (DC)

Let us consider that we have a ciphertext $C = (c_0, c_1)$ with an 8-bit payload decomposed in two nibbles. In some algorithms, it is more interesting for specific operations to work with bits. For instance, the symmetric sponge-based cipher ASCON [DEMS21] requires switching from a binary rows representation to a columns representation. Thus, we must decompose a ciphertext c into eight encryptions of bits. To do so, it is sufficient to decompose c_0 and c_1 each into four ciphertexts. That means one needs four tables: one per decomposition bit. These tables are easy to precompute as it only requires calculating for $i \in \{0, 1, \cdots, 15\}$, the LUTs corresponding to i `& 0b0001`, (i `& 0b0010`)`>>` 1, (i `& 0b0100`)`>>` 2, and (i `& 0b1000`)`>>` 3. Then, the user can perform an MVB bootstrapping using the four test-vector polynomials given by the four 1×16 tables and extract the four values. This operation is less expensive than a LUTeval call. Note that even if we now have encryptions of either 0 or 1, these ciphertexts are still manipulated with the parameters corresponding to a basis 16 encryption. This is a crucial principle for a cheap recomposition.

6.4.2 Recomposition (RC)

Once done working on a smaller basis, one should recompose their ciphertext into the initial basis to continue their computations. In our specific case of ciphertexts of basis 16 decomposed 8-bit data, that means that we want to obtain $c' = (c'_0, c'_1)$ from $c = (c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7)$ encrypting the same message m. As previously stated, the individual ciphertexts $c_0, \cdots, c_7$, even encrypting binary values, are still in the 32-value discretized torus. This simplifies the recomposition into 4-bit ciphertexts. Indeed, we have

$$c'_0 = c_0 + 2 \cdot c_1 + 2^2 \cdot c_2 + 2^3 \cdot c_3 \quad \text{and} \quad c'_1 = c_4 + 2 \cdot c_5 + 2^2 \cdot c_6 + 2^3 \cdot c_7.$$

As the multiplication by a power of 2 less than 16 will not result in an overflow, we can use a call to SimpleBoot on each c_i being multiplied. In these conditions, we can use the native TFHE TLWE addition to recompose each nibble. Still, this recomposition alone takes longer than a SHLi instruction, so a decompose/shift-for-free/recompose approach is not competitive. Hence, decomposing and recomposing ciphertexts is only efficient when many binary operations have to be performed.

7 Instructions Timings

We have fully implemented our proposed instruction set under TFHElib [CGGI16]. We summarize in Table 4 the timings obtained on our test machine (a 12th Gen Intel(R) Core(TM) i7-12700H CPU laptop with 64 Gib total system memory with an Ubuntu 22.04.2 LTS server), using only a single core.

Instr.	N_{br}	N_{ks}	**ms**	**Instr.**	N_{br}	N_{ks}	**ms**
ANDi/ORi/XORi	2	0	69	AND/OR/XOR	4	2	278
DC	2	0	81	RC	8	0	267
(U)SHLi/(U)SHRi	2	0	72	(U)SHL/(U)SHR	6	4	478
ROLi/RORi	4	0	125	ROL/ROR	9	6	714
EQi	2	0	88	EQ	6	3	393
LT(E)i/GT(E)i	2	1	126	LT(E)/GT(E)	9	5	623
(N)CDUP	3	1	159	CSEL	9	6	694
NEG/ABS	2	1	215	MIN/MAX	16	10	1176
ADDi/SUBi	2	1	137	ADD/SUB	7	4	493
ADDZ	4	2	271	MUL(M)i/DIV(4)i/MOD4i	2	1	133
MODi	3	2	267	MUL	10	6	725
MULM	32	20	2442	DIV4	21	14	1624
DIV	97	56	7711	MOD4	10	6	724
MOD	91	50	7584	(N)CDUPi	1	0	33
XOP	3	2	229	MINi/MAXi	2	1	135

Table 4: Mnemonics, blind rotations and keyswitches counts as well as execution times for our (T)FHE processor abstraction instruction set.

As a complement to Table 4, let us now illustrate the benefits of the successive optimizations that we have proposed and applied in the paper by focusing on the ADDi and ADD instructions (as representative examples of non-straightforward but not too overly complex univariate and bivariate instructions).

First, for the ADD instruction, one could start naively by trying to use the Tree-based Method relying on the basis-256 functional bootstrapping alone. This approach would require 257 blind rotates and one packing via a public functional keyswitch leading to an overall execution time of around 385 secs. Doing the same while further relying on the MVB then gets this timing down to 3 secs. Then, using our paper approach relying on a two basis-16 digits decomposition of each 8-bit data,

- Using the Tree-based Method without MVB leads to a run time of 2.32 secs for that instruction.
- With MVB (i.e. LUTeval), this number gets down to 0.52 secs.
- With the additional factorization allowed by MVLUTeval, we then obtain our final timing of 0.49 secs for the ADD instruction.

Then, regarding the ADDi instruction, one can also naively start by performing a single basis-256 bootstrapping leading to an execution time of 1.5 secs. Then, following the paper approach, we obtain the following successively refined timings:

- Using the Tree-based Method without MVB leads to 1.16 secs.

- Doing the same with MVB (LUTeval) gets that number down to 0.26 secs.
- The MVLUTeval optimization leads to 0.23 secs.
- Then, with our final optimization which factors one more simple bootstrapping (Remark 5.3, p. 13) we obtain our final timing of 0.13 secs.

Table 5 summarizes the above successive timing refinements.

Opt. level	N_{br}	N_{ks}	Timing (s)
Basis-256/TBM w/o MVB	257	1	385
Basis-256/TBM w. MVB	2	1	3
Basis-16/TBM w/o MVB	68	4	2.32
Basis-16/TBM w. MVB	8	4	0.52
Basis-16/MVLUTeval	7	4	0.49
Basis-256/SimpleBoot	1	1	1.5
Basis-16/TBM w/o MVB	34	2	1.16
Basis-16/TBM w. MVB	4	2	0.26
Basis-16/MVLUTeval	3	2	0.23
Basis-16/Rem. 5.3 (p. 5.3)	2	1	0.13

Table 5: Illustration of the benefits of the successive optimizations proposed and applied in this paper on the ADDi (lower half) and ADD (upper half) instructions.

8 From Instructions to Algorithms

To test our instruction set, we now use it to implement a number of (simple) algorithms. Note that in certain cases, it might be more efficient to directly implement these algorithms at the functional bootstrapping level. However, by analogy to a real microprocessor, that would mean coding at the micro-code rather than at the ISA level. So, in this section, we only use instructions from our set.

8.1 Testing a Few Elementary Algorithms

8.1.1 Bubble Sort

Bubble sorting consists of repeatedly comparing consecutive elements in an array and permuting them when incorrectly ordered. One way to perform the conditional swap of two array elements without resorting to an if-then-else construct can be, for example, done using MAX and MIN computations. However, it is more efficient to use GT, CDUP, NCDUP and ADDZ as done in Algo 1). To give an order of magnitude for the execution time, sorting an array of five ciphertexts encrypting 8-bit values using this "sorting in place" algorithm takes around 15 seconds. More execution timings can be found in Table 6.

8.1.2 Maximum/Minimum of an Array

As another simple example, it is easy to use our MIN and MAX homomorphic operators to find the largest or smallest element in a table, as done by Algo 2. With this algorithm, finding the maximum or minimum of an array composed of five 8-bit encrypted values takes less than 5 seconds (see Table 6).

Algorithm 1 BubbleSort

Input: A an array of n encryptions of 8-bit values
Output: A sorted from the smallest value to the largest.

for $i = n - 1$ **to** 0 **do**
  for $j = 0$ **to** $i - 1$ **do**
    $c_b \leftarrow \mathsf{GT}(A[j], A[j+1])$
    $\tilde{C} \leftarrow A[j+1]$
    $C_s \leftarrow \mathsf{CDUP}(c_b, A[j])$
    $A[j+1] \leftarrow \mathsf{NCDUP}(c_b, A[j+1])$
    $A[j+1] \leftarrow \mathsf{ADDZ}(C_s, A[j+1])$
    $C_s \leftarrow \mathsf{CDUP}(c_b, \tilde{C})$
    $A[j] \leftarrow \mathsf{NCDUP}(c_b, A[j])$
    $A[j] \leftarrow \mathsf{ADDZ}(C_s, A[j])$
return A

Algorithm 2 Maximum

Input: A an array of n encryptions of 8-bit values
Output: $\bar{C}$ a ciphertext encrypting the largest value in A

$\bar{C} \leftarrow A[0]$
for $i = 1$ **to** $n - 1$ **do**
  $\bar{C} \leftarrow \mathsf{MAX}(\bar{C}, A[i])$
return $\bar{C}$

8.1.3 Average

Thanks to our homomorphic decimal division operator, we are able to precisely compute the average of an array in fixed-point arithmetic, *including the final division*. Algo 3 gives an implementation with our instruction set. As shown in Table 6, this computation takes less than 4 seconds when tried on a five-element array.

Algorithm 3 Average

Input: A an array of n encryptions of 8-bit values
Output: $\bar{\boldsymbol{C}}$ a ciphertext encrypting the 16-bit value corresponding to the average of the table A

$C_a \leftarrow A[0]$
for $i = 1$ **to** $n - 1$ **do**
  $C_a \leftarrow \mathsf{ADD}(C_a, A[i])$
$C_i \leftarrow \mathsf{DIVi}(C_a, n)$
$C_d \leftarrow \mathsf{DIV\_DECi}(C_a, n)$
return $\bar{\boldsymbol{C}} = (C_i, C_d)$

8.1.4 Array Dereferencing and Assignement

Note that dereferencing an array of 256 (or less) *cleartext* values (with an encrypted index) is just an evaluation of our MVLUTeval operator. Further optimizations can be made on a case-by-case basis, for example, if the array to be dereferenced contains fewer than 256 values. It is

also feasible to dereference an array of 256 (or less) *encrypted* values with an encrypted index. Indeed, we can use a modified MVLUTeval running directly on encrypted test polynomials, as in the second level of the evaluation of the tree-based method (Section 3.3.3), where we rotate a new *encrypted* test polynomial by an encrypted index. Dereferencing arrays with more than 256 elements is also possible but requires a slightly more complex machinery that we do not detail (indeed, as 256 is the size of our plaintext domain we have tools to bootstrap over a 256-element table straightforwardly, but when there are more than 256 elements in the table, a single tree-based bootstrapping is not enough). Lastly, we can also obtain an operator for assigning an array of 256 (or less) encrypted values, still with an encrypted index. That is to say, given an encrypted table tab of size n, an encrypted index $C_i = (c_{i_0}, c_{i_1}) \in [\![i_0]\!] \times [\![i_1]\!]$ and an encrypted value $C_V = (c_{v_0}, c_{v_1}) \in [\![v_0]\!] \times [\![v_1]\!]$, the operation affects the value $V = 16v_0 + v_1$ to $\mathsf{tab}[16i_0 + i_1]$. See Algo 4. As seen in Table 6, the sequential evaluation of this operator on an array of five 8-bit encrypted inputs takes 4.45 seconds. Note that this approach to array dereferencing is *not* competitive with PIR approaches running over RLWE schemes. However, these latter approaches are, by intent, only able to execute PIR requests very efficiently and do not claim to achieve more than that.

Algorithm 4 Assignment

Input: A an array of n encryptions of 8-bit values, an encrypted index C_i and an encrypted value C_V

Output: A modified at index $16i_0 + i_1$

for $j = 0$ **to** $n - 1$ **do**
  $(0, c_b) \leftarrow \mathsf{EQI}(C_i, j)$
  $C_0 \leftarrow \mathsf{CDUP}(c_b; C_V)$
  $C_1 \leftarrow \mathsf{CDUP}(c_b; A[j])$
  $A[j] \leftarrow \mathsf{ADDZ}(C_0, C_1)$
return A

8.1.5 Squares Sum

As another simple algorithm, let us consider a Sum of Squares computation. It is a rather simple algorithm. However, taking a glimpse at Section 9, it already induces a number of gates large enough such that the Boolean circuit approaches are no more competitive with ours. By contrast, with our instruction set, it is (also) straightforward to implement (see Algo 5) and more efficient. Indeed, as seen in Table 6, the sequential evaluation of this operator on an array of five 8-bit encrypted inputs takes 5.66 seconds when the best Boolean circuit-based approach takes 7.6 secs (Table 8).

8.1.6 AES

Because the algorithm may be efficiently expressed by means of LUT, AES is easily implemented with our approach leading a sequential execution time of 259 seconds, as was in fact implicitly done in [TCBS23b], which is competitive with the order of magnitude in the state of the art.

Algorithm 5 SquaresSum

Input: A an array of n encryptions of 8-bit values
Output: C the encryption of the sum of the squares of the n values of A

$C_s \leftarrow \mathsf{MUL}(A[0], A[0])$
for $j = 1$ **to** $n-1$ **do**
  $C_t \leftarrow \mathsf{MUL}(A[j], A[j])$
  $C_s \leftarrow \mathsf{ADD}(C_s, C_t)$
return C_s

8.1.7 Loops

For completion, we highlight a technique to perform (encrypted) data dependant loop termination when a bound B is known on the total number of iterations. Let S denote the state of a program, then a statement of form "$\mathsf{while}\ c(S)\ \mathsf{do}\ S \coloneqq f(S)$" can be rewritten as "$\mathsf{for}\ 0 \le i < B\ \mathsf{do}\ \mathsf{if}\ c(S)\ \mathsf{then}\ S \coloneqq S\ \mathsf{else}\ S \coloneqq f(S)$". In essence, that latter form computes a fixed point after condition $c(S)$ reaches a true value, and the inner if-then-else statement can then be done via a CSEL instruction.

8.2 Evaluation of an Elementary Neuron

We now turn to the homomorphic evaluation of an elementary neuron, as usually found in convolutional neural networks. Our simple neuron has two encrypted fixed-precision inputs representing encryptions of numbers, $\boldsymbol{F_1}$ and $\boldsymbol{F_2}$, in $[-1, 1]$ (each over 16 bits as in Sect. 6.2) and one encrypted fixed-precision output of the same form. *We emphasize that the output of our neuron can be fed to another one, enabling the evaluation of larger networks over encrypted data.* From an operational viewpoint, the two encrypted inputs are first multiplied by fixed precision weights in $[-1, 1]$ ($\boldsymbol{W_1}$ and $\boldsymbol{W_2}$, respectively), which may either be cleartexts or ciphertexts. The sum of these products is then fed into an activation function, in this case, the sigmoid, noted σ (which takes an encrypted fixed-precision value as input and evaluates the sigmoid at that point). In summary, specified over cleartext value, we have to evaluate

$$\mathsf{neuron}(\boldsymbol{F_1}, \boldsymbol{F_2}) = \sigma(\boldsymbol{F_1} \cdot \boldsymbol{W_1} + \boldsymbol{F_2} \cdot \boldsymbol{W_2}).$$

Let $\boldsymbol{C_{F_1}} = (c_0, c_1, c_2, c_3) \in [\![h]\!] \times [\![l]\!] \times [\![o]\!] \times [\![k]\!]$, (meaning, as in Sect. 6.2, that $\boldsymbol{C_{F_1}}$ encrypts the value $\boldsymbol{F_1} = 16h + l + \frac{16o+k}{256}$) and $\boldsymbol{C_{F_2}} = (c'_0, c'_1, c'_2, c'_3) \in [\![h']\!] \times [\![l']\!] \times [\![o']\!] \times [\![k']\!]$. This way, we have to compute two cleartext-ciphertext decimal multiplications, one homomorphic decimal addition, and the homomorphic evaluation of the sigmoid.

The most complicated part of that computation is the homomorphic evaluation of the sigmoid, taking as input a ciphertext corresponding to a 16-bit fixed-point arithmetic value. To do so, we evaluate a discretized sigmoid $\tilde{\sigma}((i, j))$ on several non overlapping intervals $(-\infty, -6)$, $[-6, -5)$, ..., $[5, 6)$, $[6, \infty)$ as

$$\tilde{\sigma}((i,j)) = \begin{cases} (0,0) & \text{if } i < -6, \\ (0, \mathsf{tab\_sig}_i[j]) & \text{if } i \in \{-6, -5, ..., 4, 5\}, \\ (1,0) & \text{if } i \ge 6, \end{cases}$$

with $\mathsf{tab\_sig}_i[j] = \sigma(i + j/256)$ and then test the input to select the appropriate value among these. However, because we do not have any way to branch on conditions over encrypted

data, each of the above (mutually exclusive yet collectively exhaustive) possibilities must be computed, multiplied by an encrypted boolean, and then xored to obtain the final result. It follows that, for a ciphertext $\boldsymbol{C} = (c_0, c_1, c_2, c_3) \in [\![h]\!] \times [\![l]\!] \times [\![o]\!] \times [\![k]\!]$, we compute the evaluation of the sigmoid SIG by computing:

$$\bar{\boldsymbol{C}} = (\bar{C}, \tilde{C}) = (\mathsf{GTEi}((c_0, c_1), 6), \bigoplus_{i \in \{-6, \cdots, 5\}} \mathsf{CDUP}(\mathsf{EQi}((c_0, c_1), i), \mathsf{SIGLUT}^{(i)}((c_2, c_3))))$$

with $\bar{C}$ and $\tilde{C}$, the respective encryptions of the integer and decimal parts of the result. In the above equation, the $\oplus$ operator corresponds to multiple calls to our XOR[10] instruction and $\mathsf{SIGLUT}^{(i)}$ is the homomorphic evaluation of the LUT corresponding to the decimal values of $\sigma\left(i + \frac{16o+k}{256}\right)$. Note that from an instruction set perspective, $\mathsf{SIGLUT}^{(i)}$ can be performed by means of the XOP (user-defined) univariate instruction discussed in Sect. 5.3 using the above twelve $\mathsf{tab\_sig}_i$ tables. This implementation of the sigmoid takes about 10 seconds to compute. With less precise encrypted inputs and outputs, homomorphic sigmoid evaluation can be less costly [TCBS23a], but here, we prioritize accuracy and the ability to feed a neuron output into another neuron without additional conversion over faster execution. Based on this, we have been able to evaluate one neuron in about 16 seconds, so the evaluation of the sigmoid alone represents two-thirds of that cost. For comparison, we have also implemented a homomorphic Heaviside function that operates on encrypted inputs representing 16-bit fixed-point arithmetic values. Using this much simpler function, we can get the execution timing from 16 seconds down to under 7 seconds. See Table 6.

Table 6: Execution times (in seconds) of different homomorphic algorithms on arrays of size n, and expecting timings depending on n.

Algorithm	$n = 2$	$n = 5$	$n = 10$	Expected timings
BubbleSort	1.51	15.22	68.47	$\frac{n(n-1)}{2} \times (4 \times 0.160 + 2 \times 0.270 + 0.390)$
Minimum/Maximum	1.20	4.71	11.14	$(n-1) \times 1.176$
Average	0.78	2.36	4.97	$(n-1) \times 0.493 + 2 \times 0.133$
ArrayAssignment	1.37	4.45	6.88	$n \times (0.088 + 2 \times 0.159 + 0.271)$
SquaresSum	2.12	5.66	12.30	$n \times 0.725 + (n-1) \times 0.493$
	Times			
Neuron with Sigmoid	15.42	-	-	-
Neuron with Heaviside	6.67	-	-	-

9 Comparison with Other Approaches

In this section, we compare the performance of our approach to the following projects: Juliet [GMT24], E3 [EOH+18], Cingulata [CDS15a], and Concrete [Zam22]. We aim to compare the execution timings with ours on the set of algorithms from Section 8. An average timing is computed on a hundred samples for each framework and algorithm. All the results are summarized in Table 8.

[10] Note that this sum can also be computed with our ADDZ instruction.

Juliet [GMT24] Juliet[11] is a general-purpose homomorphic computation framework. It provides C++ functions corresponding to instructions implemented as Boolean circuits over ciphertexts using TFHElib. Juliet also supports GPU acceleration, but as we perform single-thread CPU computations, we have not activated this feature for fair comparison. Juliet also gives a relatively small set of around 20 instructions (working over encrypted data); it is then relevant to compare our approaches on the instructions we have in common:

- ADD, SUB, MUL, MOD
- AND, OR
- EQ, GT, LT, GTE, LTE (these instructions are all packed in COMP in Juliet)
- SHL and SHR

Additionally, many of our instructions are not supported in Juliet, particularly DIV, MOD as well as SHL and SHR (variants for shifting an encrypted value by an *encrypted* offset). Table 7 shows that with the same parameters for TFHE, our approach is, on average, 74% faster on unitary instruction execution.

Operation	This work	Juliet
AND	0.278	0.268
OR	0.278	0.254
ADD	0.493	1.012
SUB	0.493	1.231
MUL	0.725	5.334
EQ/COMP	0.393	2.195

Table 7: Average (over 100 runs) execution timings (in seconds) of different homomorphic instructions on encrypted inputs using Juliet and the present work's approach.

Cingulata [CDS15a, ACS20] Cingulata[12], formerly known as Armadillo [CDS15b], is a toolchain and run-time environment (RTE) for implementing applications running over HE. Cingulata provides high-level abstractions and tools to facilitate the implementation and execution of applications running over encrypted data. Cingulata also includes many working examples of programs. In Cingulata, programs are expressed in high-level C++ and automatically turned in optimized Boolean circuit form. Its runtime environment then performs the homomorphic execution over the selected FHE scheme, which, for this work, has been the gate-based variant of TFHE (Cingulata relies on the same TFHElib that we are also using for TFHE implementation). Additionally, even if it is not supported natively, we have implemented the homomorphic integer division given in 5.4.2 using Cingulata for further comparisons (see Table 9).

As shown in Table 8, Cingulata sometimes outperforms our approach: for algorithms Maximum and BubbleSort, it is faster by an average factor of 60%, as for the SquaresSum and Sigmoid algorithms, our approach is more efficient by an average factor of 70%. It turns out that Cingulata performs very well on homomorphic calculations, which leads to small Boolean circuits (which it executes with a faster bootstrapping). In contrast, our

[11] https://github.com/TrustworthyComputing/Juliet
[12] https://github.com/CEA-LIST/Cingulata

approach is faster when the size of the underlying Boolean circuit increases, i.e., when the ratio between the Boolean circuit size and the number of basis 16 operations is above the time ratio between basis 16 and basis 2 bootstrappings.

E3 [EOH+18] E3 (Encrypt-Everything-Everywhere)[13] is a compilation framework similar in spirit to Cingulata. Also, E3 supports bridging: homomorphic evaluation is performed by mixing both arithmetic and binary circuits, which speed up the computation in certain cases. E3, however, does not allow easy setting of the parameters for TFHE and presently uses the default TFHElib parameters, achieving only 118 bits of security (this gives E3 a slight advantage over our approach, which uses the larger fine-tuned parameters discussed in Section 4). To the best of our knowledge, E3 is the only other framework that supports homomorphic division (hence, we can compare the two approaches when computing the homomorphic average of an array). Note that in Table 2, for the `Average` algorithm, we perform a homomorphic decimal division with our framework, and only an integer division with E3. Like Cingulata, E3 outperforms our approach for the same algorithms. During our tests, Cingulata consistently outperformed E3 (except on the Average algorithm, on which our approach outperforms both E3 and Cingulata). As Sigmoid was complex to implement, we only implemented it for the most competitive approach to ours: Cingulata. (see Table 8).

Concrete [Zam22] Concrete [14] by Zama is also an FHE compiler. It is built on an LLVM-based compiler that can transform functions running over plaintexts into a mix of boolean circuits for so-called leveled operations (addition, multiplication, linear application) and LUTs (that they call TLU) for nonlinear operations that will eventually run over their custom implementation of TFHE. Its Python interface makes it easy to use: we write functions over plaintexts, compile them, and run them over ciphertexts in a single Python script. As E3, Concrete does not let the user control or even access the parameters of the TFHE scheme, meaning that we lack a complete visibility to compare their parameters with ours (Section 4). However, as both parameter sets achieve 128-bit security, we assume that the comparison remains fair.

As stated in Concrete's documentation, the tool is most efficient for computations that can be performed using leveled (in their terminology) operations rather than LUTs. This is confirmed in Table 8 where Concrete is competitive on the SquaresSum algorithm.

Algorithm	This work	Juliet	E3	Cingulata	Concrete
Maximum	4.74	7.91	3.13	**1.87**	21.52
BubbleSort	15.22	31.62	12.70	**6.18**	78.07
SquaresSum	**5.66**	25.10	8.82	8.40	7.69
Average	**2.36**	-	3.57	10.31	-
Sigmoid	**8.98**	-	-	969	-

Table 8: Average (over 100 runs) execution timings (in seconds) of different homomorphic algorithms on arrays of 5 encrypted inputs using various approaches. Entries marked with a '-' are so due to lack of support of a unitary operation (e.g., division for Average).

[13] https://github.com/momalab/e3
[14] https://github.com/zama-ai/concrete

Concluding Remarks The rationale for proposing an 8-bit instruction set (working over encryptions of two nibbles) stems from the facts that TFHE can only handle small plaintext domains without inducing huge parameters and that the computational cost of TFHE bootstrapping depends heavily on the plaintext size (as shown in Table 1). So, proceeding gate-by-gate over encryptions of bits (the FHE compilers approach) means performing large numbers of FHE operations with a fast bootstrapping (≈7 ms), while proceeding as we do requires performing comparatively fewer operations with a more costly bootstrapping (≈29 ms). The question is, therefore, whether this extra bootstrapping cost can be amortized in the instruction set approach, leading to better performances. Our experimental results in Tables 8 and 9 show that our approach amortizes this high bootstrapping cost and outperforms the other state-of-the-art compilers. This is, for example, the case for the SquaresSum algoritm (between ×1.3 and ×4.4 better), which uses MUL, an operation that requires a larger number of Boolean gates. This is further exemplified by the more complex DIVi and DIV instructions (respectively ×50 and ×1.5 faster) and the Sigmoid (×100 faster). Still, Tables 8 and 9 indeed show that the Boolean circuit-oriented approach is more efficient only for algorithms that do not suffer from a significant expansion when represented as a Boolean circuit. The above results mean that there is a significant space in which our approach outperforms the Boolean circuit approach implemented by the FHE compilers.

	This work			Cingulata	
Algorithm	N_{br}	N_{ks}	Time(s)	N_{br}'	Time(s)
DIVi	2	1	**0.13**	700	7.38
DIV	97	56	**7.71**	1100	11.17
ADD	7	4	0.49	35	**0.36**
MUL	10	6	**0.72**	183	2.14
CSEL	9	6	0.69	0	**0.001**
Maximum	64	40	4.74	184	**1.87**
BubbleSort	260	110	15.22	620	**6.18**
SquaresSum	78	46	**5.66**	821	8.40
Average	32	18	**2.36**	1000	10.31
Sigmoid	156	60	**8.98**	106652	969

Table 9: Average (over 100 runs) execution timings and number of bootstrappings of different homomorphic algorithms on arrays of 5 encrypted inputs (when applicable) using Cingulata and our approach. *Note that the top part of the above table provides a comparison of our approach with one relying on "standard TFHE gates" (as Cingulata exactly implements this approach with optimized Boolean circuits) over a set of representative instructions (implemented as small unitary programs in Cingulata).*

10 Conclusion and Perspectives

In this paper, we have essentially shown that a very limited set of functional bootstrapping patterns is both versatile and optimal to build a complete conventional-looking assembly language for manipulating (T)FHE encryptions of 8-bit data. In terms of perspectives, this reveals several functional bootstrapping operators of increasing complexity which may be appropriate targets for further works on advanced software optimizations or hardware

implementations in an intent, e.g., to provide a wide range of higher level instructions to the user while maintaining a small number of hardware operators (also leveraging on the fact that TFHE needs smaller parameters compared to the RLWE schemes). Indeed, our approach would directly benefit from further efficiency improvements in the baseline TFHE bootstrapping but also in the higher-level LUTeval or MVLUTeval operators. Beyond this, the approach can also benefit from an ability to run several such primitives in parallel, ideally by exploiting the low-level SIMD instructions offered by modern processors or dedicated HW.

Another important perspective is to further investigate several values for the bootstrapping error probability to consider the recent attacks in [CSBB24, CCP+24]. Indeed, our parameters achieve "only" a 2^{-40} bootstrapping error probability. Although parameters have been proposed in [CSBB24] for a 2^{-128} bootstrapping error probability, showing a 20% overhead in the baseline bootstrapping, they are valid only for $B = 2$. Finding a parameter set for basis $B = 16$ achieving such a low probability remains challenging (due to the necessary increase in polynomial degree and ciphertext modulus), and in that regime, basis 4 might be the optimal choice. So achieving immunity against these recent attacks may, therefore, have an impact that remains to be studied in depth.

A Complete Instruction Listing and Timings

Instr.	Description	N_{br}	N_{ks}	Exec. (ms)
ABS	Returns an encryption of the absolute value of an encrypted input.	2	1	215
ADD/SUB	Performs the homomorphic addition (or subtraction) of two encryptions of 8-bit values and returns an 8-bit encrypted result.	7	4	493
ADDi/SUBi	Performs the homomorphic addition (or subtraction) of one 8-bit plaintext to a ciphertext encrypting an 8-bit value and returns the 8-bit encrypted result.	2	1	137
ADDZ	Perform the carryless addition of two 8-bit encrypted inputs. Returns an 8-bit encrypted value.	4	2	271
AND	Computes the logical AND of two 8-bit payload ciphertexts.	4	2	278
ANDi	Computes the logical AND of an 8-bit payload ciphertext and an 8-bit plaintext.	2	0	69
(N)CDUP	Given an encrypted Boolean c_b and an 8-bit payload ciphertext C, returns an encryption of $c_b \times C$.	3	1	159
CSEL	Given an encrypted Boolean c_b and two 8-bit payload ciphertexts C and C', returns an encryption of $c_b \times C + (1 - c_b) \times C'$.	9	6	694

Instr.	Description	N_{br}	N_{ks}	Exec. (ms)
DC	Decomposition of an 8-bit payload ciphertext into eight ciphertexts encrypting binary values.	2	0	81
(U)DIV4	Calculates the Euclidean division of an 8-bit payload ciphertext by the encryption of a nibble and returns encrypted the quotient.	21	14	1624
(U)DIV(4)i	Calculates the Euclidean division of an 8-bit payload ciphertext by an 8-bit (or 4-bit) plaintext and returns the encrypted quotient.	2	1	133
(U)DIV	Calculates the Euclidean division of an 8-bit payload ciphertext by another 8-bit payload ciphertext and returns the encrypted quotient.	97	56	7711
EQ	Equality test: homomorphically compares two 8-bit payload ciphertexts and returns an encrypted boolean corresponding to the evaluation.	6	3	393
EQi	Equality test: homomorphically compares one 8-bit payload ciphertext with one 8-bit plaintext and returns an encrypted boolean corresponding to the evaluation.	2	0	88
GT(E)/LT(E)	Greater Than (or Equal to)/Less Than (or Equal to): compares two 8-bit payload ciphertexts and returns an encrypted boolean corresponding to the evaluation.	9	5	623
GT(E)i/LT(E)i	Greater Than (or Equal to)/Less Than (or Equal to): compares one 8-bit payload ciphertext with one 8-bit plaintext and returns an encrypted boolean corresponding to the evaluation.	2	1	127
MAX/MIN	Homomorphically computes the maximum (or minimum) of two 8-bit payload ciphertexts and returns the encrypted result.	16	10	1176
MAXi/MINi	Homomorphically computes the maximum (or minimum) of an 8-bit payload ciphertext and an 8-bit plaintext and returns the encrypted result.	2	1	133
MOD4	Homomorphically calculates the Euclidean division of an 8-bit ciphertext by a 4-bit ciphertext and returns the remainder.	10	6	724
MOD4i	Homomorphically calculates the Euclidean division of an 8-bit payload ciphertext by a 4-bit plaintext and returns the encryption of the remainder.	2	1	133

Instr.	Description	N_{br}	N_{ks}	Exec. (ms)
MOD	Homomorphically calculates the Euclidean division of an 8-bit payload ciphertext by another 8-bit payload ciphertext and returns the encryption of the remainder.	91	50	7584
MODi	Homomorphically calculates the Euclidean division of an 8-bit payload ciphertext by an 8-bit plaintext and returns the encryption of the remainder.	3	2	267
MUL	Multiplies two 8-bit payload ciphertexts and returns the encrypted result (only returns the result modulo 256).	10	6	725
MUL(M)i	Multiplies one 8-bit payload plaintext with an 8-bit payload ciphertext and returns the result on 8 bits.	2	1	133
MULM	Multiplies two 8-bit payload ciphertexts and returns the overhead result on 8-bits.	32	20	2442
NEG	Returns the negation of the input on a signed 8-bit payload ciphertext.	2	1	215
OR	Computes the logical OR of two 8-bit payload ciphertexts.	4	2	278
ORi	Computes the logical OR of an 8-bit payload ciphertext and an 8-bit plaintext.	2	0	69
RC	Recomposition of eight binary ciphertexts into two 4-bit payload ciphertexts encoding one 8-bit payload ciphertext.	8	0	267
ROL/ROR	Rotates an 8-bit payload ciphertext to the left (or right) by an 8-bit payload encrypted index and returns the rotated ciphertext.	9	6	714
ROLi/RORi	Rotates an 8-bit payload ciphertext to the left (or right) by an 8-bit payload plaintext and returns the rotated ciphertext.	4	0	125
(U)SHL/(U)SHR	Shifts an 8-bit payload ciphertext to the left (or right) by an 8-bit encrypted index and returns the shifted ciphertext.	6	4	478
(U)SHLi/(U)SHRi	Shifts an 8-bit payload ciphertext to the left (or right) by an 8-bit plaintext index and returns the shifted ciphertext.	2	0	72
TZR	Test Zero: Homomorphically tests if an 8-bit payload ciphertext is an encryption of zero.	2	0	88
XOP	User's defined operator.	3	2	229
XOR	Computes the logical XOR of two 8-bit paylaod ciphertexts.	4	2	278
XORi	Computes the logical XOR of an 8-bit paylaod ciphertext and an 8-bit plaintext.	2	0	69

B Instruction Set Implementation Details

B.1 Univariate Operations

Cleartext-ciphertext operations are also called *univariate* operations, as they only take *one* encrypted input. Such operations can be cleartext-ciphertext multiplications in GF(256), cleartext-ciphertext additions in GF(256), cleartext-ciphertext comparisons, ... In such cases, the generic use of our LUT evaluation tools is as follows.

In order to homomorphically compute the evaluation of a univariate function $f : \mathcal{M} \to \mathcal{M}$ on an 8-bit encrypted input $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$, we have to create basis 16 tables from tab_f, the basis 256 LookUp Table corresponding to f. Indeed, we need one table $\mathsf{tab}_{\text{MSN}}$ to compute the MSN of $f(M) = f(16h + l)$ and one $\mathsf{tab}_{\text{LSN}}$ to compute its LSN part.

Such tables are easily defined by $\mathsf{tab}_{\text{MSN}}[i] = \lfloor \frac{\mathsf{tab}_f[i]}{16} \rfloor$ and $\mathsf{tab}_{\text{LSN}}[i] = \mathsf{tab}_f[i]\%16$ for $i \in \{0, \cdots, 255\}$. Note that these tables can be efficiently generated on the fly. In fact, compared to the cost of bootstrapping, this operation is almost free of charge and offers a good time-memory compromise.

Then the naive way to evaluate f is to first compute $\bar{c_0} = \mathsf{LUTeval}(c_0, c_1; \mathsf{tab}_{\text{MSN}})$, which gives us the encryption of the MSN part of $f(M)$. Then we can similarly obtain the encryption of the LSN part of $f(M)$ by computing $\bar{c_1} = \mathsf{LUTeval}(c_0, c_1; \mathsf{tab}_{\text{LSN}})$. Thus, we obtain $\bar{C} = (\bar{c_0}, \bar{c_1}) \in [\![\bar{h}]\!] \times [\![\bar{l}]\!]$, such that $f(M) = 16\bar{h} + \bar{l}$.

It is important to see that as the calls to $\mathsf{LUTeval}$ are applied on the same encrypted inputs, they can be factorized in a single call to $\mathsf{MVLUTeval}$, giving us an *optimized* execution of the homomorphic evaluation of f. In the rest of this Section, we only present such optimized versions of our operators.

Following the same logic, we provide a user-defined operation XOP allowing one to homomorphically evaluate any univariate function on an 8-bit encrypted input of his choice.

XOP

// takes a ciphertext $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and a LUT tab_f
// $\mathsf{tab}_{\text{MSN}}$ and $\mathsf{tab}_{\text{LSN}}$ are created on the fly depending on tab_f
$(\bar{c_0}, \bar{c_1}) = \mathsf{MVLUTeval}(c_0; c_1, c_1; \mathsf{tab}_{\text{MSN}}, \mathsf{tab}_{\text{LSN}})$

Almost all univariate instructions can be implemented using the structure of the optimized XOP operation. However, this is often not optimal. For instance, we here further detail the cleartext-ciphertext addition (ADDi).

In the case of addition modulo 256 of $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ with an 8-bit plaintext $T = 16u + v \in \mathcal{M}$, we have to create the tables corresponding to the function

$$\begin{array}{rccc} f & : & \mathcal{M} & \to & \mathcal{M} \\ & & M & \mapsto & M + T \end{array}$$

The one giving the most significant nibble of computation is $\mathsf{tab\_add}_{\text{MSN}}$ such that $\mathsf{tab\_add}_{\text{MSN}}[i] = \lfloor \frac{i+T}{16} \rfloor$ for $i \in \{0, \cdots, 255\}$. Similarly, the table giving the least significant nibble of the computation is $\mathsf{tab\_add}_{\text{LSN}}$ such that $\mathsf{tab\_add}_{\text{LSN}}[i] = (i + T) \pmod{16}$ for $i \in \{0, \cdots, 255\}$.

Using these tables (that can be computed on the fly), we can execute the ADDi operation following the pattern of XOP. But, if we twist our $\mathsf{MVLUTeval}$ tool just a bit (following Remark 5.3), we can obtain an even more optimized version of these operators. To compute the LSN part of the homomorphic addition of $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and $T = 16u + v$, the table $\mathsf{tab}_{\text{LSN}}$ such that for $i \in \{0, \cdots, 15\}$, $\mathsf{tab}_{\text{LSN}}[i] = i + v\%16$ is sufficient. That means that a call to

SimpleBoot on c_1 can give us the LSN part of the computation (instead of a call to LUTeval). To factorize this bootstrapping with the one implied by the computation of $\bar{c_0}$ we modify the MVLUTeval instruction: we create $\mathsf{MVLUTeval}_2$ so that the MVB method is used with selector c_1 on $\mathsf{tab}_{\text{MSN}}$ and $\mathsf{tab}_{\text{LSN}}$. Then we apply the extractions: we get 17 new ciphertexts. Sixteen of these ciphertexts (corresponding to the first part of the evaluation of $\mathsf{tab}_{\text{MSN}}$ are used to proceed to the final bootstrapping (giving an encryption of $\mathsf{tab}_{\text{MSN}}[16h+l]$), and the last ciphertext is $\bar{c_1} \in [\![l+u \pmod{16}]\!]$. The generic implementation of such optimized versions is as follows:

ADDi, SUBi, LT(E)i, GT(E)i, MULi, MULMi, DIV(4)i, MOD(4)i, MINi, MAXi, NEG, ABS, ...

// takes a ciphertext $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and an 8-bit immediate value T
// $\mathsf{tab}_{\text{MSN}}$ and $\mathsf{tab}_{\text{LSN}}$ are created on the fly
$(\bar{c_0}, \bar{c_1}) = \mathsf{MVLUTeval}_2(c_1; c_0, c_0; \mathsf{tab}_{\text{MSN}}, \mathsf{tab}_{\text{LSN}})$

This optimization saves us one bootstrapping and one TLWE to TRLWE keyswitch.

B.1.1 Shifts ((U)SHLi, (U)SHRi)

In logical (unsigned) shift, zeros are inserted to replace displaced bits as well in signed representation as in unsigned representation. In the specific case of cleartext-ciphertext logical shift, one wants to shift an 8-bit encrypted input $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ by a plaintext $K \in \mathcal{M}$. To be as efficient as possible and avoid as many unnecessary FHE calculations as possible, we must proceed by considering different cases based on the value of the plaintext index. Indeed, since we work with messages of only 8 bits, an offset index greater than or equal to 8 would not require any homomorphic calculation: it would suffice to return encryption of 0, that is to say $\bar{C} = (\bar{c_0}, \bar{c_1}) \in [\![0]\!] \times [\![0]\!]$. Similarly, if the offset index is $K = 16 \cdot 0 + 4$, it is sufficient to simply return $\bar{C} = (\bar{c_0}, \bar{c_1}) \in [\![l]\!] \times [\![0]\!]$ or $\bar{C} = (\bar{c_0}, \bar{c_1}) \in [\![0]\!] \times [\![h]\!]$ depending on the direction of the shift. Finally, if the index is $K = 16 \cdot 0 + 0$, then the output is the unmodified input ($\bar{C} = C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$). Now, if $K \in \{1, 2, 3, 5, 6, 7\}$, we have to use LUTs to compute the new MSN and LSN of the ciphertext. As the offset index K is known, the tables are easy to compute. For instance for the unsigned shift to the left (USHLi), for $i \in \{0, \cdots, 15\}$, we have $\mathsf{tab\_ushli}_{\text{MSN}}[i] = ((i \texttt{<<} K) \texttt{ >> } 4) \texttt{ \&0xf}$ and $\mathsf{tab\_ushli}_{\text{LSN}}[i] = (i \texttt{<<} K) \texttt{ \&0xf}$. This way, we can compute $\mathsf{USHLi}(C, K)$ by means of SimpleBoot and native TFHE addition LweAdd:

USHLi

// takes a ciphertext $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and an 8-bit immediate value K
$\bar{c_1} = \mathsf{SimpleBoot}(c_1, \mathsf{tab\_ushli}_{\text{LSN}})$
$c_t = \mathsf{SimpleBoot}(c_0, \mathsf{tab\_ushli}_{\text{LSN}})$
$c_u = \mathsf{SimpleBoot}(c_1, \mathsf{tab\_ushli}_{\text{MSN}})$
$\bar{c_0} = \mathsf{LweAdd}(c_t, c_u)$

The computations of $\bar{c_1}$ and c_u can be factorized with an MVB.

The final result is $\bar{C} = (\bar{c_0}, \bar{c_1}) = \mathsf{USHLi}(C, T) \in [\![\mathsf{tab\_shli}_{\text{MSN}}[16h+l]]\!] \times [\![\mathsf{tab\_shli}_{\text{LSN}}[16h+l]]\!]$. Here we can use the native TFHE addition for two reasons. First, the noise of ciphertext after a classic bootstrapping is small enough (regarding our parameters set designed especially

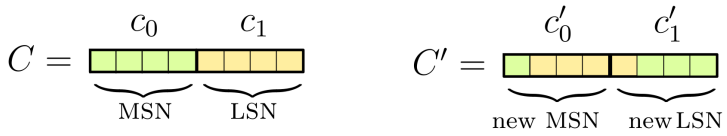

Figure 2: New MSN and LSN of an 8-bit encrypted value rotated by three to the left ($C' = \mathsf{UROLi}(C, 3)$).

to work for ciphertexts obtained following a tree-based method), and second, this is a carryless addition, so there will not be any carry to propagate, meaning that the result stays an encryption of a nibble value. The cleartext-ciphertext logical shift to the right and arithmetic shift to the left are computed in a similar way. In the case of signed right arithmetic (signed) shift (SHRi), the sign bit is replicated to fill in all the vacant positions. The tables to use are thus the following:

- $\mathsf{tab\_shri}_{\text{LSN}}$ such that for $i \in \{0, \cdots, 7\}$, $\mathsf{tab\_shri}_{\text{LSN}}[i] = (i\texttt{<<}4)\texttt{>>}K$ `&` `0xf`
  and for $i \in \{8, \cdots, 15\}$, $\mathsf{tab\_shri}_{\text{LSN}}[i] = ((i\texttt{\&0xf0})\texttt{<<}4)\texttt{>>}K$ `&` `0xf`
- $\mathsf{tab\_shri}_{\text{MSN}}$ such that for $i \in \{0, \cdots, 7\}$, $\mathsf{tab\_shri}_{\text{MSN}}[i] = (i\texttt{>>}K)$ `&` `0xf`
  and for $i \in \{8, \cdots, 15\}$, $\mathsf{tab\_shri}_{\text{MSN}}[i] = ((i\texttt{\&0xf0})\texttt{>>}K)$ `&` `0xf`
- $\mathsf{tab\_ushri}_{\text{MSN}}$ such that for $i \in \{0, \cdots, 15\}$, $\mathsf{tab\_ushri}_{\text{MSN}}[i] = (i\texttt{>>}K)$ `&` `0xf`

SHRi

// takes a ciphertext $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and an 8-bit immediate value K
$\bar{c_0} = \mathsf{SimpleBoot}(c_0, \mathsf{tab\_shri}_{\text{MSN}})$
$c_t = \mathsf{SimpleBoot}(c_1, \mathsf{tab\_ushri}_{\text{MSN}})$
$c_u = \mathsf{SimpleBoot}(c_0, \mathsf{tab\_shri}_{\text{LSN}})$
$\bar{c_1} = \mathsf{LweAdd}(c_t, c_u)$

The computations of $\bar{c_0}$ and c_u can be factorized with an MVB.

B.1.2 Rotations (ROLi, RORi)

With rotations, bits that are "shifted out" are reinserted at the end or beginning of the word, depending on the shift direction (see Figure 2). To implement cleartext-ciphertext rotations, it is sufficient to use the same tables as for the unsigned shifts. For instance, here is the left rotation (ROLi).

ROLi

// takes a ciphertext $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and an 8-bit immediate value K
$c_t = \mathsf{SimpleBoot}(c_1, \mathsf{tab\_ushli}_{\mathrm{MSN}})$
$c_u = \mathsf{SimpleBoot}(c_0, \mathsf{tab\_ushli}_{\mathrm{LSN}})$
$\bar{c_1} = \mathsf{LweAdd}(c_t, c_u)$
$c_t = \mathsf{SimpleBoot}(c_0, \mathsf{tab\_ushli}_{\mathrm{MSN}})$
$c_u = \mathsf{SimpleBoot}(c_1, \mathsf{tab\_ushli}_{\mathrm{LSN}})$
$\bar{c_0} = \mathsf{LweAdd}(c_t, c_u)$

The computations of c_t and c_u can be factorized with an MVB.

B.1.3 Univariate Bitwise Operations

All univariate bitwise operators can be homomorphically evaluated using the same tools LUTeval and MVLUTeval as before. But in this specific case, it is not the most efficient way. Indeed, regarding bitwise operands, MSN and LSN parts are independent. Thus, two simple bootstrappings are sufficient to compute the resulting MSN and LSN, there is no need for a tree-based method. For instance, to homomorphically compute the bitwise operation XORi of a ciphertext $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ with an 8-bit immediate value $T = 16u + v$, we only need to create two 16-elements tables. The first one, corresponding to the evaluation of the MSN is $\mathsf{tab}_{\mathsf{XORi}_u}$ such that for $j \in \{0, \cdots, 15\}$, $\mathsf{tab}_{\mathsf{XORi}_u}[j] = j \oplus u$. The second, corresponding to the LSN is $\mathsf{tab}_{\mathsf{XORi}_v}$ such that for $j \in \{0, \cdots, 15\}$, $\mathsf{tab}_{\mathsf{XORi}_v}[j] = j \oplus v$. Then, the evaluation of XORi is as follows.

XORi

// takes a ciphertext $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and an 8-bit immediate value T
// $\mathsf{tab}_{\mathrm{MSN}}$ and $\mathsf{tab}_{\mathrm{LSN}}$ are created on the fly depending on T
$\bar{c_0} = \mathsf{SimpleBoot}(c_0, \mathsf{tab}_{\mathrm{MSN}})$
$\bar{c_1} = \mathsf{SimpleBoot}(c_1, \mathsf{tab}_{\mathrm{LSN}})$

The final result is $\bar{C} = (\bar{c_0}, \bar{c_1}) = \mathsf{XORi}(C, T) \in [\![u \oplus h]\!] \times [\![v \oplus l]\!]$. All other univariate bitwise operations (including CDUPi) are similarly implemented.

B.1.4 Other Exceptions

Other cleartext-ciphertext operations that have a different structure to XOP include EQi, which tests whether a ciphertext $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ is an encryption of a plaintext $T = 16u + v$. Only one 256-element table, depending on the immediate value T and straightforward to compute, is needed. This table is tab such that for $i \in \{0, \cdots, 255\}$, $\mathsf{tab}[i] = (i == T)$. Then, the instruction is implemented as follows:

EQi

// takes a ciphertext $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and an 8-bit immediate value T
// tab is created on the fly depending on T
$\bar{c_1} = \mathsf{LUTeval}(c_0, c_1, \mathsf{tab})$

Depending on the programmer choice, $\bar{c_0}$ is either an encryption of 0, a plaintext value 0, or $\perp$. This choice is left to the programmer's discretion since the value of $\bar{c_0}$ will not be used in subsequent calculations (the boolean result is encrypted only in $\bar{c_1}$).

B.2 Bivariate Operations

Unlike univariate operators, there is no generic way of efficiently handling bivariate operators. The number of tables and calls to LUTeval and MVLUTeval will depend on the type of operation required, which is why we provide more details for numerous types of operations.

B.2.1 Bitwise Operators

For clarity's sake, let us denote $\circledast$ any bitwise operator (such as XOR, AND, NOR, etc.) or any composition of these operators. All bitwise bivariate operators can be homomorphically evaluated using one 256-element table and MVLUTeval. Indeed, the LUT table corresponding to the bitwise operator $\circledast$ is $\mathsf{tab}_\circledast$ such that for $i, j \in \{0, \cdots, 15\}$, $\mathsf{tab}_\circledast[16i + j] = i \circledast j$. Note that the coefficients of $\mathsf{tab}_\circledast$ are in $\{0, \cdots, 15\}$. Then, the homomorphic computation of the $\circledast$ bitwise operand on two ciphertexts $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$ only costs two calls to LUTeval. For instance, the XOR instruction is computed as follows.

XOR

// takes two ciphertexts $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$
$\bar{c_0} = \mathsf{LUTeval}(c_0, c'_0; \mathsf{tab}_{\mathsf{XOR}})$
$\bar{c_1} = \mathsf{LUTeval}(c_1, c'_1; \mathsf{tab}_{\mathsf{XOR}})$

The final result is $\bar{C} = (\bar{c_0}, \bar{c_1}) = \mathsf{XOR}(C, C') \in [\![h \oplus h']\!] \times [\![l \oplus l']\!]$. All other bivariate bitwise operations are similarly implemented.

B.2.2 Addition (ADD, SUB)

With 8-bit messages decomposed into two nibbles, the addition over $\mathbb{Z}_{256}$ is not completely straightforward. Indeed, to sum $M = 16h + l$ and $M' = 16h' + l'$, we first have to sum the two least significant nibbles of the messages. That is to say to compute $L = l + l' = 16u + v$. As the sum of l and l' may be greater than 15, we have to compute not only v, but also the carry u. Then we can compute $H = h + h' + u$, but in this case we do not compute the carry as we work modulo 256.
To proceed to these computations in the homomorphic domain, we thus need two tables:

- $\mathsf{tab\_add}$ which computes the LSN part of the addition of two nibbles. That is to say for $i, j \in \{0, \cdots, 15\}$, $\mathsf{tab\_add}[16i + j] = (i + j) \pmod{16}$.
- $\mathsf{add\_carry}$ which computes the carry corresponding to the addition of two nibbles. That is to say for $i, j \in \{0, \cdots, 15\}$, $\mathsf{add\_carry}[16i + j] = \lfloor \frac{i+j}{16} \rfloor$.

Using these two tables, we implement the homomorphic addition the following way:

ADD

// takes two ciphertexts $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$
$(\bar{c_1}, c_r) = \mathsf{MVLUTeval}(c_1; c'_1, c'_1; \mathsf{tab\_add}, \mathsf{add\_carry})$
$c_v = \mathsf{LUTeval}(c_0, c'_0, \mathsf{tab\_add})$
$\bar{c_0} = \mathsf{LUTeval}(c_v, c_r, \mathsf{tab\_add})$

The final result is $\bar{C} = (\bar{c_0}, \bar{c_1}) = \mathsf{ADD}(C, C') \in [\![\lfloor \frac{16h+l+16h'+l'}{16} \rfloor]\!] \times [\![l + l' \pmod{16}]\!]$. Note that the substraction modulo 256 (SUB) works similarly, only the LUTs are different.

The Case of the Addition by Zero (ADDZ) – If the user knows that at least one of the two ciphertexts $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ or $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$ is an encryption of zero, then he should use a less expensive operator than the one described above. Indeed, if one of the two ciphertexts encrypts zero, then the addition with any other ciphertext will not produce any carry. It is thus more efficient to compute such an addition as follows:

ADDZ

// takes two ciphertexts $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$
$\bar{c_0} = \mathsf{LUTeval}(c_0, c'_0, \mathsf{tab\_add})$
$\bar{c_1} = \mathsf{LUTeval}(c_1, c'_1, \mathsf{tab\_add})$

This specific operator can be needed in several cases: for instance a homomorphic array assignment (Algo 4), a bubble sort (Algo 1), or the computation of the Sigmoid (Section 8.2).

B.2.3 Multiplication (MUL(M))

The multiplication of two 8-bit messages each decomposed into two nibbles relies on the same principle as the addition: we progress nibble by nibble and propagate the carry. We note $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$ two ciphertexts respectively encrypting $T = 16h + l$ and $T' = 16h' + l'$. Then, the following relation stands for all $T, T' \in \mathcal{M}$.

$$T \times T' \;=\; (16h + l) \times (16h' + l') \;=\; 16^2 hh' \;+\; 16(hl' + lh') \;+\; ll'$$

Since we work with 8-bit messages, we do not need to compute the term in 16^2. Thus, we only have to compute $16(hl' + lh') \;+\; ll'$, which involves three multiplications and two additions over $\mathbb{Z}_{256}$. To compute the additions, we use the tables created in Section B.2.2. For the multiplication, we create two new tables:

- $\mathsf{tab\_mul}$ which computes the LSN part of the multiplication of two nibbles. That is to say for $i, j \in \{0, \cdots, 15\}$, $\mathsf{tab\_mul}[16i + j] = (i \times j) \pmod{16}$.
- $\mathsf{mul\_carry}$ which computes the carry corresponding to the addition of two nibbles. That is to say for $i, j \in \{0, \cdots, 15\}$, $\mathsf{mul\_carry}[16i + j] = \lfloor \frac{i \times j}{16} \rfloor$.

Note that we only need to compute the LSN parts of hl' and lh', as well as the LSN of the sum of these two terms (the MSN parts will be multiples of 16^2 modulo 256). For the same reason, the MSN of ll' will be added with regards only to the least significant bits of the result. Finally, multiplication can then be achieved as follows:

MUL

$(\bar{c_1}, c_t, c_u) = \mathsf{MVLUTeval}(c_1; c'_1, c'_1, c'_0; \mathsf{tab\_mul}, \mathsf{mul\_carry}, \mathsf{tab\_mul})$
$//c_t$ encrypts the carry of ll', c_u encrypts the LSN of lh'
$c_v = \mathsf{LUTeval}(c_0, c'_1; \mathsf{tab\_mul})$
$c_w = \mathsf{LUTeval}(c_u, c_v; \mathsf{tab\_add})$
$\bar{c_0} = \mathsf{LUTeval}(c_w, c_t; \mathsf{tab\_add})$

The final result is $\bar{C} = (\bar{c_0}, \bar{c_1}) = \mathsf{MUL}(C, C') \in [\![\lfloor \frac{(16h+l)\times(16h'+l')}{16} \rfloor]\!] \times [\![ll' \pmod{16}]\!]$.

To Obtain the Most Significant Byte – When we multiply two 8-bit integers, we obtain a result on 16 bits. By working with an 8-bit processor, we need a different operation than the multiplication modulo 256 to obtain the Most Significant Byte. To do so, we have no choice but to compute the whole operation without using the ADD instruction, which only works modulo 256. That is to say, to obtain the Most Significant Byte of the multiplication of a ciphertext $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ with a ciphertext $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$, we have to homomorphically compute

$$T \times T' \;=\; (16h + l) \times (16h' + l') \;=\; 16^2 hh' \;+\; 16(hl' + lh') \;+\; ll'.$$

with $T = 16h + l$ and $T' = 16h' + l'$ the corresponding two 8-bit plaintexts. The only unnecessary computation is the one giving the LSN of the term ll'. Indeed, all the other operations need to be computed in order to propagate the carry. We can summarize this method as follows:

MULM

$(c_u, c_v, c_w, c_x) = \mathsf{MVLUTeval}(c_0; c'_0, c'_0, c'_1, c'_1; \mathsf{tab\_mul}, \mathsf{tab\_mul}, \mathsf{tab\_mul}, \mathsf{tab\_mul})$
$(c_t, c_y, c_z) = \mathsf{MVLUTeval}(c_1; c'_0, c'_0, c'_1; \mathsf{mul\_carry}, \mathsf{tab\_mul}, \mathsf{mul\_carry})$
$(c_x, c_y) = \mathsf{MVLUTeval}(c_x; c_y, c_y; \mathsf{add\_carry}, \mathsf{tab\_add})$
$(c_t, c_w) = \mathsf{MVLUTeval}(c_w; c_t, c_t; \mathsf{add\_carry}, \mathsf{tab\_add})$
$(c_c, c_s) = \mathsf{MVLUTeval}(c_x; c_w, c_w; \mathsf{add\_carry}, \mathsf{tab\_add})$
$c_z = \mathsf{LUTeval}(c_y, c_z, \mathsf{add\_carry})$
$(c_s, c_z) = \mathsf{MVLUTeval}(c_s; c_z, c_z, \mathsf{add\_carry}, \mathsf{tab\_add})$
$c_c = \mathsf{LUTeval}(c_c, c_s, \mathsf{add\_carry})$
$(c_0, \bar{c_1}) = \mathsf{ADD}((c_u, c_v), (c_c, c_z))$

The result is $\bar{C} = (\bar{c_0}, \bar{c_1}) = \mathsf{MULM}(C, C') \in [\![\lfloor (16h + l) \times (16h' + l')/4096 \rfloor]\!] \times [\![\lfloor (16h + l) \times (16h' + l')/256 \rfloor \pmod{16}]\!]$.

B.2.4 Minimum and Maximum (MIN, MAX)

Without loss of generality, we describe here the homomorphic computation of the minimum of two ciphertexts $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$. The computation of the maximum is very similar and can easily be inferred from that of the minimum.
To evaluate the MIN instruction with our MVLUTeval and LUTeval tools, several tables are needed:

- tab_min, such that for $i, j \in \{0, 1, \cdots, 15\}$, $\mathsf{tab\_min}[16i + j] = \min(i, j)$
- is_inf such that for $i, j \in \{0, 1, \cdots, 15\}$, $\mathsf{is\_inf}[16i + j] = (i < j)$.

- $\mathsf{is\_sup}$ such that for $i, j \in \{0, 1, \cdots, 15\}$, $\mathsf{is\_sup}[16i + j] = (i > j)$.
- $\mathsf{is\_eq}$ such that for $i, j \in \{0, 1, \cdots, 15\}$, $\mathsf{is\_eq}[16i + j] = (i == j)$.

Then, we can obtain $c_x = \mathsf{LUTeval}(c_0, c'_0, \mathsf{is\_inf}) \in [\![h < h']\!] \subset \mathcal{C}$, $c_y = \mathsf{LUTeval}(c_0, c'_0, \mathsf{is\_sup}) \in [\![h > h']\!] \subset \mathcal{C}$, and $c_z = \mathsf{LUTeval}(c_0, c'_0, \mathsf{is\_eq}) \in [\![h == h']\!] \subset \mathcal{C}$ with a factorized call to

$$\mathsf{MVLUTeval}(c_0; c'_0, c'_0, c'_0; \mathsf{is\_inf}, \mathsf{is\_sup}, \mathsf{is\_eq}).$$

Then, the result of the evaluation of the minimum of two ciphertexts C and C' is

$$\bar{C} = \mathsf{MIN}(C, C') = (\mathsf{LUTeval}(c_0, c'_0, \mathsf{min\_tab}),\ c_x \cdot c_1 \ +\ c_y \cdot c'_1 \ +\ c_z \cdot \mathsf{LUTeval}(c_1, c'_1, \mathsf{min\_tab})).$$

Note that the required homomorphic multiplications always involve a ciphertext that encrypts 0 or 1 (because c_x, c_y, and c_z are encryptions of booleans). This means that these multiplications will not produce any carry. It is thus sufficient to only compute the LSN result of these multiplications. The same goes for the required additions: only one of the three terms will be positive, and the others will encrypt zero.

MIN

$(c_x, c_y, c_z, \bar{c_0}) = \mathsf{MVLUTeval}(c_0; c'_0, c'_0, c'_0, c'_0; \mathsf{is\_inf}, \mathsf{is\_sup}, \mathsf{is\_eq}, \mathsf{tab\_min})$
$(c_y, \tilde{c_1}) = \mathsf{MVLUTeval}(c'_1; c_y, c_1, \mathsf{tab\_mul}, \mathsf{tab\_min})$
$c_z = \mathsf{LUTeval}(c_z, \tilde{c_1}, \mathsf{tab\_mul})$
$c_x = \mathsf{LUTeval}(c_x, c_1, \mathsf{tab\_mul})$
$c_{xy} = \mathsf{LUTeval}(c_x, c_y, \mathsf{tab\_add})$
$\bar{c_1} = \mathsf{LUTeval}(c_{xy}, c_z, \mathsf{tab\_add})$

B.2.5 Division by a 4-bit Ciphertext (DIV4)

Following the presentation of the homomorphic division in Section 5.4.2, we use the following tables to implement the division by a ciphertext $C_k \in [\![0]\!] \times [\![k]\!]$:

- $\mathsf{div}_{\mathrm{MSN}_1}$ such that for $i, j \in \{0, \cdots, 15\}$, $\mathsf{div}_{\mathrm{MSN}_1}[16i + j] = \lfloor \frac{16i}{j} \rfloor / 16$
- $\mathsf{div}_{\mathrm{MSN}_2}$ such that for $i, j \in \{0, \cdots, 15\}$, $\mathsf{div}_{\mathrm{MSN}_2}[16i + j] = \lfloor \frac{16i}{j} \rfloor \% 16$
- $\mathsf{div}_{\mathrm{LSN}}$ such that for $i, j \in \{0, \cdots, 15\}$, $\mathsf{div}_{\mathrm{LSN}}[16i + j] = \lfloor \frac{i}{j} \rfloor$
- $\mathsf{mod}_{\mathrm{LSN}}$ such that for $i, j \in \{0, \cdots, 15\}$, $\mathsf{mod}_{\mathrm{LSN}}[16i + j] = i \% j$
- $\mathsf{mod}_{\mathrm{MSN}}$ such that for $i, j \in \{0, \cdots, 15\}$, $\mathsf{mod}_{\mathrm{MSN}}[16i + j] = 16i \% j$

DIV4

$(c_u, c_v, c_w, c_x, c_y) = \mathsf{MVLUTeval}(c_k; c_0, c_0, c_1, c_0, c_1; \mathsf{div}_{\mathrm{MSN}_1}, \mathsf{div}_{\mathrm{MSN}_2}, \mathsf{div}_{\mathrm{LSN}}, \mathsf{mod}_{\mathrm{MSN}}, \mathsf{mod}_{\mathrm{LSN}})$
$(c_t, c_z) = \mathsf{MVLUTeval}(c_x; c_y, c_y, \mathsf{add\_carry}, \mathsf{tab\_add})$
$c_t = \mathsf{LUTeval}(c_t, c_z, \mathsf{tab\_add})$
$c_s = \mathsf{LUTeval}(c_t, c_w, \mathsf{tab\_add})$
$(c_s, \bar{c_1}) = \mathsf{MVLUTeval}(c_v; c_s, c_s; \mathsf{add\_carry}, \mathsf{tab\_add})$
$\bar{c_0} = \mathsf{LUTeval}(c_u, c_s, \mathsf{tab\_add})$

Details on division by an 8-bit encrypted value $C'_k = (c_{k_0}, c_{k_1}) \in [\![k_0]\!] \times [\![k_1]\!]$ are given in Section 5.4.2, as well as the pseudo code of DIV.

B.2.6 Modulo

(a) MOD4

Let us say that we want to compute the homomorphic modulo of one 8-bit encrypted value $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ corresponding to the encryption of $T = 16h + l$ by the 4-bit encrypted value $c_k \in [\![k]\!]$ (note that if $c_k \in [\![0]\!]$, the instruction should return an error, but we choose to return encryptions of 0). This means we have to homomorphically compute

$$16h + l \pmod{k} = (16h \pmod{k} + l \pmod{k}) \pmod{k}.$$

But, to do so, computing $16h \pmod{k} + l \pmod{k}$ is not sufficient and could result in a ciphertext encrypting a value superior to k. For instance, given $k = 15$, $h = 14$ and $l = 14$ we obtain $16h \pmod{k} + l \pmod{k} = 28 > k$. This example also highlights that $16h \pmod{k} + l \pmod{k}$ may require two nibbles to hold the result.

However, we have:

$$\begin{aligned} 16h \pmod{k} + l \pmod{k} &\leq k - 1 + k - 1 \\ &\leq 2k - 2 \end{aligned}$$

If we note $16h \pmod{k} + l \pmod{k} = 16h' + l'$ with $h', l' \in \{0, ..., 15\}$, then

$$16h' \pmod{k} + l' \pmod{k} \leq k - 1$$

Indeed, if $h' = 0$, then $16h' \pmod{k} + l' \pmod{k} = l' \pmod{k} < k$, and if $h' > 0$, then $16h' > k$ so $16h' \pmod{k} \leq 16h' - k$ and $16h' \pmod{k} + l' \pmod{k} < 2k - 2 - k = k - 2$. Thus computing $16h' \pmod{k} + l' \pmod{k}$ gives us a ciphertext encrypting a value inferior to k, that is the smallest positive representative of the class of $T \pmod{k}$.

To implement it with our LUT evaluation tools, we need two tables, mod16 and mod, defined by $\mathsf{mod16}[16i + j] = 16i\%j$ and $\mathsf{mod}[16i + j] = i\%j$ with $i, j \in \{0, 1, \cdots, 15\}$.

MOD4

$(c_r, c_u) = \mathsf{MVLUTeval}(c_k; c_0, c_1; \mathsf{mod16}, \mathsf{mod})$
$(c_t, c_z) = \mathsf{MVLUTeval}(c_r; c_u, c_u, \mathsf{add\_carry}, \mathsf{tab\_add})$
$c_s = \mathsf{MVLUTeval}(c_k; c_t, c_z, \mathsf{mod16}, \mathsf{mod})$
$\bar{c_1} = \mathsf{LUTeval}(c_t, c_s, \mathsf{tab\_add})$

Depending on the user's choice, $\bar{c_0}$ is either an encryption of 0, a plaintext value 0, or $\perp$.

(b) MOD

Now, let us say that a user wants to compute the homomorphic modulo of a ciphertext $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ by another ciphertext $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$ respectively encrypting the 8-bit values $T = 16h + l$ and $T' = 16h' + l'$. This computation is similar to the computation of DIV, we recall here how it can be computed.

MOD

$c_d = \mathsf{LUTeval}(c_0, c'_1, \mathsf{div}_{\text{LSN}})$
$c_s = \mathsf{LUTeval}(c'_0, c_d, \mathsf{tab})$
$c_t = \mathsf{LUTeval}(c_s, c'_1, \mathsf{mul\_carry})$
$c_d = \mathsf{LUTeval}(c_0, c_t, \mathsf{tab\_sub})$
// Computation of the MSN part of the result is over, we now compute the LSN
$\tilde{C} = (c_d, c_1)$
For $i = 3$ to 0
  $C_m = (c_{l0}, c_{l1}) = \mathsf{SHLi}(C', i)$
  $c_g = \mathsf{GTE}(\tilde{C}, C_m)$
  $c_b = \mathsf{LUTeval}(c'_0, c_g, \mathsf{tab\_and\_mulm\_zero})$
  $C_s = \mathsf{MVLUTeval}(c_b; c_{l0}, c_{l1}; \mathsf{tab\_mul}, \mathsf{tab\_tab\_mul})$
  $\tilde{C} = \mathsf{SUB}(\tilde{C}, C_s)$

With $\tilde{C}$ the final result. tab is such that for $i, j \in \{0, \cdots, 15\}$, $\mathsf{tab}[16i + j] = (i == 0) \times j$. Other tables are defined in Section 5.4.2.

B.2.7 Comparisons

In this section, we discuss the case of several ciphertext-ciphertext comparisons that return an encrypted boolean. Depending on the programmer choice, $\bar{c_0}$ is either an encryption of 0, a plaintext value 0, or $\perp$. This choice is left to the programmer's discretion since the value of $\bar{c_0}$ will not be used in subsequent calculations (the boolean result is encrypted in $\bar{c_1}$).

(a) Equality Test of Two Ciphertext (EQ)
To compute the homomorphic equality test of two ciphertexts $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$ two methods are available. The first one is to homomorphically compute the subtraction of one of the two 8-bit ciphertexts by the other and then homomorphically test if the result ciphertext is an encryption of zero. We have already created all the tools required for these operations (see previous sections). So, it is only a question of reusing what has already been made rather than creating new tables. The other method involves creating a new table for evaluating comparisons of 4-bit ciphertexts. Indeed, we have to precompute a 16×16 equality table $\mathsf{tab\_eq}$ such that for $i, j \in \{0, \cdots, 15\}$, $\mathsf{tab\_eq}[16i + j] = (i == j)$. Then we implement the EQ operator the following way:

EQ

$c_t = \mathsf{LUTeval}(c_0, c'_0, \mathsf{tab\_eq})$
$c_s = \mathsf{LUTeval}(c_1, c'_1, \mathsf{tab\_eq})$
$\bar{c_1} = \mathsf{LUTeval}(c_t, c_s, \mathsf{tab\_mul})$

(b) Greater/Less Than (GT(E)/LT(E))
Given two 8-bit ciphertexts $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$ respectively encrypting $T = 16u + v$ and $T' = 16u' + v'$, we want to homomorphically determine either if one of them is greater or smaller than the other. To do so, we have to create new tables for evaluating comparisons of 4-bit ciphertexts. Indeed, for instance, for the "Greater Than" operator, we have $T > T'$ if and only if $h > h'$ or $h == h'$ and $l > l'$. So we need a table that gives the evaluation "Greater Than" and one that gives the evaluation "Equal

To". The first one we call tab_greater_than, and is computed such that for $i, j \in \{0, \cdots, 15\}$, tab_greater_than$[16i + j]$ = $(i > j)$. Similarly, as seen in the previous section, we create tab_eq such that for $i, j \in \{0, \cdots, 15\}$, tab_eq$[16i + j]$ = $(i == j)$. Instructions GTE, LT and LTE are implemented in a similar way.

GT

(c_t, c_s) = MVLUTeval$(c_0; c'_0, c'_0;$ tab_greater_than, tab_eq$)$
c_u = LUTeval$(c_1, c'_1,$ tab_greater_than$)$
c_b = LUTeval$(c_s, c_u,$ tab_mul$)$
$\bar{c_1}$ = LUTeval$(c_t, c_b,$ tab_mul$)$

B.2.8 Rotations and Shifts

(a) Shifts ((U)SHL, (U)SHR)

In the case of a shift of an 8-bit encrypted input $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ by an encrypted 4-bit index $c_k \in [\![k]\!]$,as the offset index is encrypted, one cannot proceed by considering different cases based on the value of the offset to optimize the FHE calculations. We thus have to create LUTs encoding the result of the shift and evaluate them with our LUTeval and MVLUTeval tools to be able to compute the operation. Without loss of generality, let us say that we want to compute a homomorphic arithmetic signed shift to the right SHR. Several tables are needed:

- shr_{MSN} such that for $i, j \in \{0, \cdots, 15\}$, $\text{shr}_{\text{MSN}}[16i + j]$ = $(i$>>$j)$&0xf
- ashr_{MSN} such that for $i \in \{0, \cdots, 7\}$ and $j \in \{0, \cdots, 15\}$, $\text{ashr}_{\text{MSN}}[16i + j] = \text{shr}_{\text{MSN}}[16i + j]$ and for $i \in \{8, \cdots, 15\}$ and $j \in \{0, \cdots, 15\}$, $\text{ashr}_{\text{MSN}}[16i + j]$ = $((i$&0xf0$)$>>$j)$&0xf
- ashr_{LSN} such that for $i \in \{0, \cdots, 7\}$ and $j \in \{0, \cdots, 15\}$, $\text{ashr}_{\text{LSN}}[16i + j]$ = $((i$<<4$)$>>$j)$&0xf and for $i \in \{8, \cdots, 15\}$ and $j \in \{0, \cdots, 15\}$, $\text{ashr}_{\text{LSN}}[16i + j]$ = $(((i$&0xf0$)$ << 4$)$>>$j)$&0xf

Then, SHR can be implemented as follows:

SHR

$(\bar{c_0}, c_s, c_u)$ = MVLUTeval$(c_k; c_0, c_0; c_1 \text{ashr}_{\text{MSN}}, \text{ashr}_{\text{LSN}}, \text{shr}_{\text{MSN}})$
$\bar{c_1}$ = LUTeval$(c_u, c_s,$ tab_add$)$

Other shifts are computed in a similar way with their corresponding tables.

(b) Rotations (ROL,ROR)

Rotations are computed using the same logic as the logical shift. The only difference is that instead of inserting zeros (or signed bit) to fill the space left by the bits that have been pushed out, we recover these bits and reinsert them. The LUTs are identical to the one needed for logical shift. ROR instruction is computed in a symmetric way.

ROL

(c_t, c_s, c_u, c_x) = MVLUTeval$(c_k; c_0, c_0, c_1; c_1; \text{shl}_{\text{MSN}}, \text{shl}_{\text{LSN}}, \text{shl}_{\text{MSN}}, \text{shl}_{\text{LSN}})$
$\bar{c_0}$ = LUTeval$(c_t, c_x,$ tab_add$)$
$\bar{c_1}$ = LUTeval$(c_u, c_s,$ tab_add$)$

B.2.9 Conditional Assignment

We propose two conditional assignment operators that we call CDUP and NCDUP. They each take two ciphertexts $C = (c_0, c_1) \in [\![h]\!] \times [\![l]\!]$ and $C' = (c'_0, c'_1) \in [\![h']\!] \times [\![l']\!]$ as inputs. The first one is an encryption of a boolean value (meaning $h = 0$ and $l \in \{0, 1\}$), and the second one can be an encryption of any 8-bit value. We define these instructions so that $\mathsf{CDUP}(C, C') = \bar{C} \in [\![h \times l']\!] \times [\![l \times l']\!]$ and $\mathsf{NCDUP}(C, C') = \bar{C} \in [\![h \times (1 - l')]\!] \times [\![l \times (1 - l')]\!]$. To implement these, we either need the tab_mul table used to compute the LSN of the multiplication of two encrypted nibbles (CDUP) or a modified tab_mul called tab_mul_spe such that for $i, j \in \{0, \cdots, 15\}$, $\mathsf{tab\_mul\_spe}[16i + j] = (j == 0) \times i$ (NCDUP).

(N)CDUP

//Depending on the instruction to be computed, we use tab which is either tab_mul
//or tab_mul_spe
$(\bar{c_0}, \bar{c_1}) = \mathsf{MVLUTeval}(c'_1; c_0, c_1; \mathsf{tab}, \mathsf{tab})$

C Additional Background on TFHE Bootstrapping

C.1 Further Details on TFHE Bootstrapping

TFHE bootstrapping relies on three building blocks:

- BlindRotate: rotates a plaintext polynomial m encrypted with a TRLWE sample (a, b) and the secret key k by a position p encrypted with a TLWE sample (a', b') with the secret key s. It takes as inputs: the TRLWE ciphertext $(a, b) \in [\![m]\!]_k$, a rescaled and rounded vector of $(a', b') \in [\![p]\!]_s$ represented by $(a'_1, \cdots, a'_n, a'_{n+1} = b')$ where $\forall i,\ a'_i \in \mathbb{Z}_{2N}$, and n TRGSW ciphertexts encrypting $(s_1, \cdots, s_n)$ where $\forall i,\ s_i \in \mathbb{B}$. It returns a TRLWE ciphertext $(a", b") \in ([\![X^{\langle a,s\rangle - b} \cdot m]\!]_k)$.
- TLWESampleExtract: takes as inputs both a TRLWE sample $c \in [\![m]\!]_k$ and a position $p \in [\![0, N[\![$, and returns a TLWE ciphertext $c' \in [\![m_p]\!]_k$ where m_p is the p^{th} coefficient of the polynomial m.
- PublicFunctionalKeyswitch: transforms a set of p ciphertexts $c_i \in [\![m_i]\!]_k$ into the resulting TRLWE ciphertext $c' \in [\![f(m_1, \cdots, m_p)]\!]_s$, where $f()$ is a public linear morphism from $\mathbb{T}^p$ to $\mathbb{T}_N[X]$. Note that $N = 1$ when keyswitching to a TLWE ciphertext. This algorithm requires 2 parameters: a decomposition basis B_{KS} and a precision t.

TFHE specifies a gate bootstrapping to reduce the noise level of a TLWE sample that encrypts the result of a boolean gate evaluation on two ciphertexts, each of them encrypting a binary input. TFHE gate bootstrapping steps are summarized in Algorithm 6. The step 1 consists in selecting a value $\hat{m} \in \mathbb{T}$ which will serve later for setting the coefficients of the test polynomial *testv* (in step 3). The step 2 rescales the components of the input ciphertext c as elements of $\mathbb{Z}_{2N}$. The step 3 defines the test polynomial *testv*. Note that for all $p \in [\![0, 2N[\![$, the constant term of $testv \cdot X^p$ is $\hat{m}$ if $p \in]\!]\frac{N}{2}, \frac{3N}{2}]\!]$ and $-\hat{m}$ otherwise. The step 4 returns an accumulator $ACC \in [\![testv \cdot X^{\langle \bar{a}, s\rangle - \bar{b}}]\!]_{s'}$. Indeed, the constant term of ACC is $-\hat{m}$ if $c \in [\![0]\!]_s$, or $\hat{m}$ if $c \in [\![1]\!]_s$. Then, step 5 creates a new ciphertext $\overline{c}$ by extracting the constant term of ACC and adding to it $(0, \hat{m})$. That is, $\overline{c}$ either encrypts 0 if $c \in [\![0]\!]_s$, or m if $c \in [\![1]\!]_s$ (By choosing $m = \frac{1}{2}$, we get a fresh encryption of 1).

Algorithm 6 TFHE gate bootstrapping

Require: a constant $m \in \mathbb{T}$, a TLWE sample $c = (a, b) \in [\![x \cdot \frac{1}{2}]\!]_s$ with $x \in \mathbb{B}$, a bootstrapping key $BK_{s \to s'} = (BK_i \in [\![s_i]\!]_{S'})_{i \in [\![1,n]\!]}$ where BK_i is a TRGSW sample of s_i with the key S'; the TRLWE interpretation of a secret key s',

Ensure: a TLWE sample $\overline{c} \in [\![x.m]\!]_s$

1: Let $\hat{m} = \frac{1}{2}m \in \mathbb{T}$ (pick one of the two possible values)
2: Let $\bar{b} = \lfloor 2Nb \rceil$ and $\bar{a}_i = \lfloor 2Na_i \rceil \in \mathbb{Z}, \forall i \in [\![1, n]\!]$
3: Let $testv := (1 + X + \cdots + X^{N-1}) \cdot X^{\frac{N}{2}} \cdot \hat{m} \in \mathbb{T}_N[X]$
4: $ACC \leftarrow \mathsf{BlindRotate}((0, testv), (\bar{a}_1, \cdots, \bar{a}_n, \bar{b}), (BK_1, \cdots, BK_n))$
5: $\overline{c} = (0, \hat{m}) + \mathsf{TLWESampleExtract}(ACC)$
6: return $\mathsf{PublicFunctionalKeyswitch}_{\mathsf{s' \to s}}(\overline{c})$

C.2 Further Details on Multi-Value Bootstrapping

Multi-Value Bootstrapping (MVB) [CIM18] refers to the method for evaluating k different LUTs on a single input with a single bootstrapping. MVB factors the test polynomial P_{f_i} associated with the function f_i into a product of two polynomials v_0 and v_i, where v_0 is a common factor to all P_{f_i}. In practice, we have:

$$(1 + X + \cdots + X^{N-1}) \cdot (1 - X) \equiv 2 \bmod (X^N + 1)$$

Now by writing P_{f_i} in the form $P_{f_i} = \sum_{j=0}^{N-1} \alpha_{i,j} X^j$ with $\alpha_{i,j} \in \mathbb{Z}$, we get from the previous equation:

$$\begin{aligned} P_{f_i} &= \frac{1}{2} \cdot (1 + X + \cdots + X^{N-1}) \cdot (1 - X) \cdot P_{f_i} \bmod (X^N + 1) \\ &= v_0 \cdot v_i \bmod (X^N + 1) \end{aligned}$$

where:

$$\begin{aligned} v_0 &= \frac{1}{2} \cdot (1 + X + \cdots + X^{N-1}) \\ v_i &= \alpha_{i,0} + \alpha_{i,N-1} + (\alpha_{i,1} - \alpha_{i,0}) \cdot X + \cdots + (\alpha_{i,N-1} - \alpha_{i,N-2}) \cdot X^{N-1} \end{aligned}$$

This factorization allows computing many LUTs using a unique bootstrapping. Indeed, it is enough to initialize the test polynomial $testv$ with the value of v_0 during bootstrapping. Then, after the `BlindRotate` operation, one has to multiply the obtained ACC by each v_i corresponding to the LUT of f_i to get ACC_i. Figure 3 illustrates the advantage of this method. This optimization reduces the number of bootstrappings required for an operation and, thus, the overall computation time. The MVB technique can be applied on the first "round" of a tree-based method evaluation, as several bootstrappings are performed on different polynomials but with the same encrypted input. For instance, regarding Figure 1, instead of doing five bootstrappings to compute the evaluation of the identity function on the encrypted message $M = (1, 2)$, one can use the MVB and compute the same evaluation at the cost of only two bootstrappings (and four multiplications).

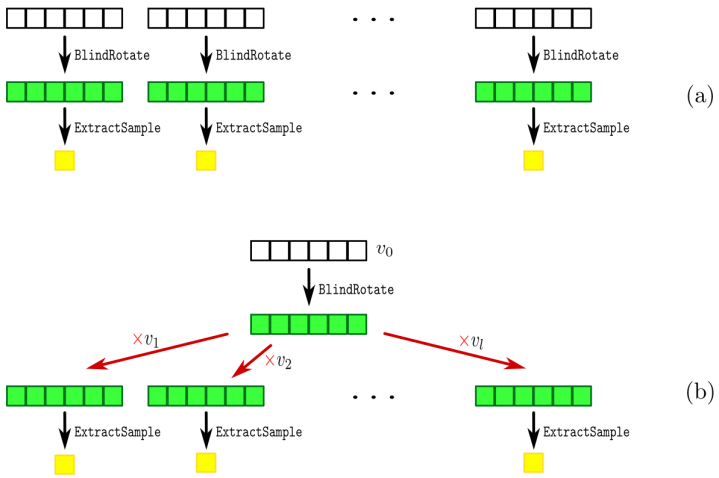

Figure 3: Illustration of the MVB optimization. (a) represents the classic method to process several bootstrapping, while (b) represents the MVB optimization. As seen here, it reduces the number of `BlindRotate` operations, which is the most expansive one of the bootstrapping.

References

[ABB+16] J. Bacelar Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying Constant-Time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[ACS20] P. Aubry, S. Carpov, and R. Sirdey. Faster homomorphic encryption is not enough: Improved heuristic for multiplicative depth minimization of boolean circuits. In *Topics in Cryptology – CT-RSA 2020*, 2020.

[BBB+] L. Bergerat, A. Boudi, Q. Bourgerie, I. Chillotti, D. Ligier, J-B. Orfila, and S. Tap. Parameter Optimization & Larger Precision for (T)FHE. https://www.zama.ai/post/parameter-optimization-and-larger-precision-for-tfhe.

[BBB+23] L. Bergerat, A. Boudi, Q. Bourgerie, I. Chillotti, D. Ligier, J-B. Orfila, and S. Tap. Parameter Optimization and Larger precision for (T)FHE. *Journal of Cryptology*, 36, 2023.

[BBS22] A.-A. Bendoukha, A. Boudguiga, and R. Sirdey. Revisiting stream-cipher-based homomorphic transciphering in the tfhe era. In *Foundations and Practice of Security*, 2022.

[BCBS23] A.-A. Bendoukha, P.-E. Clet, A. Boudguiga, and R. Sirdey. Optimized stream-cipher-based transciphering by means of functional-bootstrapping. In *Data and Applications Security and Privacy XXXVII*, 2023.

[BCCS24] A. Bondarchuk, O. Chakraborty, G. Couteau, and R. Sirdey. Downlink (t)FHE ciphertexts compression. Cryptology ePrint Archive, Paper 2024/1921, 2024.

[BDGM19] Z. Brakerski, N. Döttling, S. Garg, and G. Malavolta. Leveraging linear decryption: Rate-1 fully-homomorphic encryption and time-lock puzzles. In *TCC*, 2019.

[BMMP18] F. Bourse, M. Minelli, M. Minihold, and P. Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *CRYPTO*, 2018.

[BPR24] N. Bon, D. Pointcheval, and M. Rivain. Optimized Homomorphic Evaluation of Boolean Functions. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024.

[BPS] M. Brenner, H. Perl, and M. Smith. How practical is homomorphically encrypted program execution? an implementation and performance evaluation. In *IEEE TrustCom*.

[CBSZ23] P.-E. Clet, A. Boudguiga, R. Sirdey, and M. Zuber. 2023.

[CCF+16] A. Canteaut, S. Carpov, C. Fontaine, T. Lepoint, M. Naya-Plasencia, P. Paillier, and R. Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. In *Fast Software Encryption*, 2016.

[CCP+24] J. H. Cheon, H. Choe, A. Passelègue, D. Stehlé, and E. Suvanto. Attacks against the IND-CPAD security of exact FHE schemes. Technical Report 127, IACR ePrint, 2024.

[CDS15a] S. Carpov, P. Dubrulle, and R. Sirdey. Armadillo: a compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, 2015.

[CDS15b] S. Carpov, P. Dubrulle, and R. Sirdey. Armadillo: a compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, pages 13–19, 2015.

[CGGI16] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast Fully Homomorphic Encryption Library, August 2016. https://tfhe.github.io/tfhe/.

[CGGI19] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33, 2019.

[CGRS14] D. B. Cousins, J. Golusky, K. Rohloff, and D. Sumorok. An FPGA co-processor implementation of homomorphic encryption. In *IEEE HPEC*, 2014.

[CIM18] S. Carpov, M. Izabachène, and V. Mollimard. New techniques for Multi-value input Homomorphic Evaluation and Applications. In *Topics in Cryptology – CT-RSA 2019*, 2018.

[CJP21] I. Chillotti, M. Joye, and P. Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *Cyber Security Cryptography and Machine Learning*, 2021.

[CKK19] J.H. Cheon, D. Kim, and D. Kim. Efficient homomorphic comparison methods with optimal complexity. In *Advances in Cryptology – ASIACRYPT 2020*, 2019.

[CLOT21] I. Chillotti, D. Ligier, J-B. Orfila, and S. Tap. Improved Programmable Bootstrapping with Larger Precision and Efficient Arithmetic Circuits for TFHE. In *Advances in Cryptology – ASIACRYPT 2021*, 2021.

[CS19] A. Chatterjee and I. Sengupta. FURISC: FHE encrypted URISC design. In *Fully Homomorphic Encryption in Real World Applications*. Springer, 2019.

[CSBB24] M. Checri, R. Sirdey, A. Boudguiga, and J.-P. Bultel. On the practical cpad security of "exact" and threshold FHE schemes. In *CRYPTO*, 2024.

[DEMS21] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer. Ascon v1.2: Lightweight authenticated encryption and hashing. *Journal of Cryptology*, 34, 2021.

[EOH+18] Chielle E., Mazonka O., Gamil H., Georgios Tsoutsos N., and Maniatakos M. E3: A framework for compiling c++ programs with encrypted operands. Cryptology ePrint Archive, Report 2018/1013, 2018.

[FSF+13] S. Fau, R. Sirdey, C. Fontaine, C. Aguilar-Melchor, and G. Gogniat. Towards practical program execution over fully homomorphic encryption schemes. In *IEEE 3PGCIC*, 2013.

[GBA21] A. Guimarães, E. Borin, and D. F. Aranha. Revisiting the functional bootstrap in TFHE. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2), 2021.

[GMT24] C. Gouert, D. Mouris, and N. G. Tsoutsos. Juliet: A Configurable Processor for Computing on Encrypted Data. *IEEE Transactions on Computers*, pages 1–14, 2024.

[GN20] X. Gong and D. Negrut. Cryptoemu: An instruction set emulator for computation over ciphers. Technical Report TR-2020-10, University of Wisconsin-Madison, 2020.

[IMP18] F. Irena, D. Murphy, and S. Parameswaran. CryptoBlaze: A partially homomorphic processor with multiple instructions and non-deterministic encryption support. In *IEEE ASP-DAC*, 2018.

[IZ21] I. Iliashenko and V. Zucca. Faster homomorphic comparison operations for BGV and BFV. Cryptology ePrint Archive, Paper 2021/315, 2021.

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

[LLKN21] E. Lee, J-W. Lee, Y-Sik. Kim, and J-S. No. Optimization of homomorphic comparison algorithm on RNS-CKKS scheme. *IEEE Access*, 2021.

[PJH23] Méaux P., Park J., and V. L. Pereira H. Towards practical transciphering for FHE with setup independent of the plaintext space. *IACR Communications in Cryptology*, 2023.

[SV10] N. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *PKC*, 2010.

[TCBS23a] D. Trama, P.-E. Clet, A. Boudguiga, and R. Sirdey. Building blocks for lstm homomorphic evaluation with tfhe. In *Cyber Security, Cryptology, and Machine Learning*, 2023.

[TCBS23b] D. Trama, P.-E. Clet, A. Boudguiga, and R. Sirdey. A Homomorphic AES Evaluation in Less than 30 Seconds by Means of TFHE. WAHC, 2023.

[TM13] N. G. Tsoutsos and M. Maniatakos. Investigating the application of one instruction set computing for encrypted data computation. In *SPACE*, 2013.

[TM14] N. G. Tsoutsos and M. Maniatakos. HEROIC: Homomorphically encrypted one instruction computer. In *IEEE DATE*, 2014.

[YXS+21] Z. Yang, X. Xie, H. Shen, S. Chen, and J. Zhou. Tota: Fully homomorphic encryption with smaller parameters and stronger security. Cryptology ePrint Archive, Report 2021/1347, 2021.

[Zam22] Zama. Concrete: TFHE Compiler that converts python programs into FHE equivalent, 2022. https://github.com/zama-ai/concrete.