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

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

Université Paris-Saclay, CEA-List, Palaiseau, France
{nom.prenom}@cea.fr

Abstract

Making the most of TFHE programmable bootstrapping to evaluate functions or operators otherwise difficult 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-bits data and presents itself to the programmer by means of a conventional-looking assembly instruction set. In doing so, we provide several homomorphic LUT dereferencing operators based on variants on the tree-based method and show that they are the most efficient option for manipulating encryptions of 8-bit data (optimally represented as two base 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 then conclude the paper by testing the approach on several simple algorithms, including the execution of a neuron with a sigmoid activation function over 16-bit precision. 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 — FHE · 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 addition over $\mathbb{Z}_t$ ($t >> 2$), one has to perform many boolean operations. Because of this, work on FHE has 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 to address concrete applications,

*This work was supported by the France 2030 ANR Project ANR-22-PECY-003 SecureCompute.

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 despite a number of 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. At the other end of the spectrum stands TFHE. On the downside, TFHE is intrinsically an LWE scheme, meaning that it offers no batching (apart from 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 functionnal 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 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[1]. As one may intuit, however, the resulting instruction set is quite different from that of a usual processor as many instructions cannot be performed directly when working in the encrypted domain, and new instructions have to 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 relies heavily 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 [BPR23]. We first review these methods and select the most appropriate one along with the most appropriate 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 as a basis to build our set of 8-bit instructions.

Related works – To the best of our knowledge, previous attempts at building FHE-based virtual processors are quite 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. [BPS12][2], 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 very small 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 3 branching instructions. Each (encrypted) instruction is fetched from the encrypted memory and then homomorphically interpreted (at an

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

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

extra cost equivalent to that of 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 in the sense that 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), which then has to be emulated using explicit conditional assignments at an extra cost equivalent to that of explicitly running all branches. Also, we can then afford to have a much more complete instruction set, which is tailored to the capabilities of our modern functional bootstrapping tool. 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 furthermore likely to induce vulnerabilities in realistic deployment scenarios since TFHE is trivially insecure against a CCA(1) adversary. By opposition, we "handle" 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 without any focus on instruction sets.

Contributions – This paper's contributions are as follows:

- We define a set of functionnal bootstrapping tools and optimal parameters to manipulate encryptions of 8-bit data by means of LUT dereferencing by 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 (T)FHE 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.
- 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 or even hardware acceleration.

Paper Organization – This paper is organized as follows: Section 2 reviews the basics of the TFHE cryptosystem and gives the necessary details of the tree-based method for bootstrapping with multi-input ciphertexts and its optimization with multi-value bootstrapping. Section 3 details the

rationale for the functional bootstrapping technique and associated parameters selection. Sections 4, 5 and 6 (unitary timings) focus on our instruction set, which is then used in Section 7 to implement several algorithms. Finally, Section 8 concludes this paper with some perspectives.

2 Preliminaries

2.1 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.

2.2 The TFHE Scheme

The TFHE scheme is a fully homomorphic encryption scheme introduced in 2016 [CGGI16] and implemented in the TFHE library[3]. TFHE defines three structures to encrypt plaintexts, which we summarize below as fresh encryptions of 0:

- 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 $e \in \mathbb{T}$ 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 vectors of size 1 whose coefficient is a polynomial.
- 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 $\mathcal{M} \in \mathbb{T}$)[4]. 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 explicitly need such an operation in this paper, more details about TRGSW can be found in [CGGI16]. 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] tfhe.github.io/tfhe/

[4] 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 $\mathcal{M} = \{0, 0.25, 0.5, 0.75\}$.

2.3 TFHE Bootstrapping and Programmable Bootstrapping

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 (see [CGGI16] for details). The first operation, BlindRotate, rotates a plaintext polynomial $testv$[5] 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. $\mathsf{N_{br}}$ will simplify comparing instructions implemented with the same set of TFHE parameters.

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 so allows the implementation of a LUT of f. We refer to this bootstrapping as *programmable* or *functional*. We note that the original bootstrapping is a special 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[6] 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 $X^{\phi(c)} \mod (X^N+1)$[7]. 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 $m \in \mathcal{M} \cap [\frac{1}{2}, 1)$.

2.3.1 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 rather small set. In 2021, Guimarães *et al.* [GBA21] specified the *tree-based* and the *chaining* methods for performing functional bootstrapping over several ciphertexts. Their idea is to decompose the input message being encrypted 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 shows 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 with each cleartext polynomial and 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

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

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

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

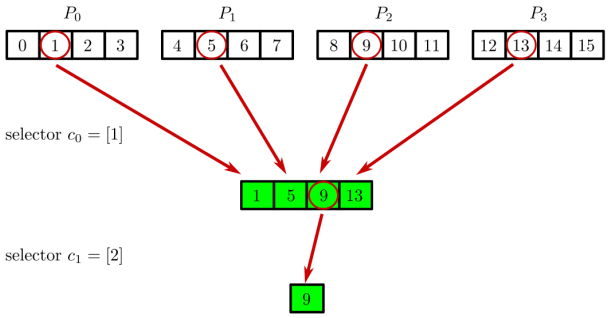

Figure 1: Illustration of the tree-based method on the identity function with decomposition in basis $B = 4$. The message is $m = 9 = 1 \cdot 4^0 + 2 \cdot 4^1$ and its corresponding encryption is $C = ([1], [2])$. Red arrows indicate bootstrapping.

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].

2.3.2 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 many 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 [CIM18]. 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.

3 Choosing the Right Toolbox

3.1 On the Choice of the Functional Bootstrapping Method

Univariate functional bootstrapping – Many works tackled the restriction of TFHE bootstrapping to the evaluation of LUTs of negacylic functions (Sect. 2.3). 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 $\mathsf{BlindRotate}$ ($\mathsf{N_{br}}\geq 2$). 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.
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 2.3.2. 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 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 growth in parameters jeopardizes the other speed improvements that could come with the chaining method compared ot the tree-based method [TCBS23b]. A recent work by Bon et al. [BPR23] proposes a method to evaluate boolean functions with several encrypted inputs with one bootstrapping. However, their method is limited to binary plaintexts that must be 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 size of the truth table of the function, which makes it difficult 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. A follow-up study presented in [BBB+], however, 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.

3.2 Optimal Basis Selection for LUT Evaluation

Decomposition basis choice – The plaintext space corresponding to 8-bit messages is the set $\mathcal{P} = \{0, 1, \cdots, 255\}$. Since we use the half-torus bootstrapping method, we have to work on a 512-element discretized torus to match such $\mathcal{P}$. This requires very large TFHE parameters leading to a very slow bootstrapping (≈1.5 secs for a single bootstrapping [TCBS23b]). As a consequence, 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 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 1 refers to the trick of computing the output of the first level of the tree with MVB instead of running B^{d-1} bootstrappings (Sect. 2.3.2). 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 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 the ones 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 $\frac{1}{10}$ in basis 8 and $\frac{1}{22}$ 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 basis 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, we can conclude that basis 16 is the optimal choice. † **LUT dereferencing operators –** Now

Table 1: Execution times of SimpleBoot, LUTeval and MVLUTeval depending on the plaintext decomposition basis. MVLUTeval* stands for an evaluation of two different LUTs, and MVLUTeval$^\diamond$ for four different LUTs.

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

that we know 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 $c \in [\![m]\!]$, SimpleBoot$(c;$ tab_16$)$ returns $c' \in [\![\text{tab\_16}[m]]\!]$. 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 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.

$\mathsf{LUTeval}(c_0, c_1;\ \mathsf{tab})$ 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 ($\mathsf{tab\_1},\cdots,\mathsf{tab\_k}$). 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 $k+1$ BlindRotate ($\mathsf{N_{br}} = k+1$) instead of $2k$ ones for computing k LUTeval, with k 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 $c'_j \in [\![\mathsf{tab\_j}\,[16m_\alpha + m_j]]\!]\ \forall j \in \{1,\cdots,k\}$. MVLUTeval can be further optimized when provided with the same table tab_j twice (or more) by computing less PublicFunctionalKeyswitch.

Table 2: Parameter sets 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 PublicFunctionalKeyswitch, q denotes the size of the used plaintext size, and ϵ is the error probability of one MVB tree-based evaluation.

basis	n	N	l	B_g	B_{KS}	t	q	ϵ	TRLWE noise	TLWE noise
4	700	1024	5	16	1024	2	8	2^{-30}	5.6×10^{-8}	1.9×10^{-5}
8	700	2048	2	2048	1024	2	16	2^{-23}	9.6×10^{-11}	1.9×10^{-5}
16	1024	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 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

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

4 An FHE-Optimized Instruction Set

4.1 Instruction Set Overview

In this paper, we propose an exhaustive set of some fifty 8-bit instructions which manipulate (T)FHE-encrypted data. Some of the provided instructions are relatively standard, but others are more specific and included because a smallernumber of homomorphic operations is 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-bits 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* (a case which occurs recurrently in several algorithmic patterns). 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 processor ISA but also 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. All 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 be able to emulate if-then-else or do-while statements with encrypted data-dependant 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 may be arbitrarily configured by the programmer.

For readability's sake and also due to space limitation, the following sections are intended only to discuss the key difficulties we had to overcome and the optimization techniques we had to consider in order to implement the full set of instructions.

4.2 Notations for Homomorphic Operator Specifications

In this work, following Sect. 3 and most particularly Sect. 3.2, 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 respectively 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. In some cases, a ciphertext C may have no MSN and is denoted as $(\perp, c_1)$. 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 instantitate in Sect. 3.2.

4.3 Implementing Univariate Instructions

In this work, univariate instructions are those that 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). Following the notation introduced in the previous section, 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[8] 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 encryption of a boolean 1-bit value (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

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

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. 4.2, 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 (overall 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 be used to 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 in order to evaluate custom conditions following (2). Table 3 provides a synthetic (yet exhaustive) list of the univariate instructions we have implemented.

[8] 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...

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), (U)ROL(i) (rotate an encrypted byte left by an encrypted 8-bit index); (U)SHR(i); (U)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); ABS (absolute value of an encrypted signed byte); NEG (returns the opposite of an encrypted signed byte); XOP; XOPN

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.

4.4 Implementing Bivariate Instructions

Bivariate Instructions Basics – We now turn to bivariate instructions, which are instructions that 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 Sect. 4.2, we have already seen how to perform bitwise instructions. As another simple 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 $\bar{C}$:

$$\begin{aligned} c_s &= \mathsf{LUTeval}(c_0, c'_0, \mathsf{tab\_add4}) \\ (\bar{c}_1, c_c) &= \mathsf{MVLUTeval}(c_1; c'_1, c'_1; \mathsf{tab\_add4}, \mathsf{tab\_carry}) \\ \bar{c}_0 &= \mathsf{LUTeval}(c_s, c_c, \mathsf{tab\_add4}). \end{aligned} \tag{3}$$

with $\mathsf{tab\_add4}[16i + j] = (i + j) \mod 16$ and $\mathsf{tab\_carry}[16i + j] = \lfloor (i + j)/16 \rfloor$.

As for the univariate case (Sect. 4.3), bivariate test instructions output only a single (encrypted) nibble with a single-bit payload, i.e., ciphertexts of the form $(\perp, c_1)$ with $c_1 \in [\![0]\!] \cup [\![1]\!]$. Quite often in FHE computations, we have to sum two encrypted values where an unknown one of them encrypts 0. On a cleartext processor, this does not make any difference, and a special instruction is usually not included for that case. When working over encrypted data, however, the lack of carry propagation means that we can save a call to MVLUTeval, (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 independant 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,

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

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 do not detail them further. See Table 3 for a full list.

An Homomorphic Division Operator – Using division as a yardstick, 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 here 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

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

Following this, Q can be naively obtained by means of 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 $\bar{c_0} = \mathsf{LUTeval}(\mathsf{SimpleBoot}(c'_0, \mathsf{tab\_is\_zero}), \mathsf{LUTeval}(c_0, c'_1; \mathsf{tab\_div}); \mathsf{tab\_mul})$. Then, to compute the LSN of the quotient, we must follow the algorithm presented in Figure 2, 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 the following way. First, we need to compute $c_s = \mathsf{LUTeval}(\bar{c_0}, c'_1; \mathsf{tab\_mul})$, and then update c_0 with the value $c_t = \mathsf{LUTeval}(c_0, c_s, \mathsf{tab\_sub})$.

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. †

5 Other Types of Ciphertexts

5.1 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 she is using signed or unsigned representation to be able to correctly interpret the decrypted messages. 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,

$C_t = (c_t, c_1)$
$c_q \in [\![0]\!]$
for $i = 3$ to 0
$C_m = (c_{m_0}, c_{m_1}) = \mathsf{SHLi}(C', i)$ // Shift Left by a cleartext index
$c_g = \mathsf{GTE}(C_t, C_m)$ // Greater Than or Equal
$c_b = \mathsf{LUTeval}(c'_0, c_g, \mathsf{tab\_and\_mulm\_zero})$
$C_s = \mathsf{MVLUTeval}(c_b; c_{m_0}, c_{m_1}; \mathsf{tab\_mul\_lsn}, \mathsf{tab\_mul\_lsn})$
$C_t = \mathsf{SUB}(C_t, C_s)$
$c_q = \mathsf{LUTeval}(c_q, c_b, \mathsf{tab\_add\_}q_i)$

Figure 2: Division pseudo-code to obtain the LSN of $\bar{C}$. $\mathsf{tab\_and\_mulm\_zero}$ is a 256-element table with $\mathsf{tab\_and\_mulm\_zero}[16k + j] = (((k << i) >> 4) == 0)\&(j == 1)$, 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. $\mathsf{tab\_add\_}q_i$ is a 256-element table such that for $k, j \in \{0, \cdots, 15\}$, $\mathsf{tab\_add\_}q_i[16k + j] = k + (j \neq 0) \cdot 2^i$ that we use to add the new q_i to c_q. Finally, note that for $i = 0$, SHLi does nothing.

multiplications, shifts,... and to perform new operations such as the negation NEG or the absolute value ABS. For many operators, these operations are very similar to their unsigned variants, we do not detail them further.

5.2 Support for Fixed-point Arithmetic

We can also apply this paper 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 semantics 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 new functions to be implemented, such as decimal division or a fixed-precision sigmoid (Section 7). As an example, let us consider the decimal division by a cleartext 8-bit value d (assuming unsigned input semantic for simplicity sake), an operation which 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 $\mathsf{BlindRotate}$, we then compute

$$(\bar{c}_0, \bar{c}_1, \tilde{c}_0, \tilde{c}_1) = \mathsf{MVLUTeval}(c_0; c_1, c_1, c_1, c_1; \mathsf{tab\_int\_msn}, \mathsf{tab\_int\_lsn}, \mathsf{tab\_dec\_msn}, \mathsf{tab\_int\_lsn})$$

5.3 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 useful 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 (and similarly so on the downlink).

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 thus 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, this leads to an expansion factor of only $\frac{32}{4} = 8$, which is reasonable by "FHE standards".

Downlink output data transmission – Remark that none of the above two approaches are applicable to reduce the overhead of encrypted results transmission from the server to the client. Indeed, transciphering allows to convert 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 somehow transmitted. 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. Let us thus consider that we want to assemble $K = 2L$ TLWE results into one or more TRLWE ciphertexts. More precisely, if the server has to send back K encrypted bytes, i.e., $2L$ TLWE ciphertexts. Then the server assembles up to n TLWE samples into a single TRLWE sample by means of 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 $K \bmod n = 0$, 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 ($t = 16$). So, the downlink expansion factor is "only" twice that of the uplink (asymptotically). When, $K \bmod n = r > 0$, 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].

5.4 Bit decomposition and recomposition

Decomposition – 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 certain operations to work with bits. For instance, the symmetric block cipher ASCON [DEMS21] requires switching from a binary rows representation to a columns representation. We thus need to 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 0s and 1s, these ciphertexts are still manipulated with the parameters corresponding to a basis 16 encryption. This is a key principle for a cheap recomposition.

Recomposition – Once done working over the smaller basis, one should recompose his or her ciphertext into the initial basis to continue his or her 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 0s and 1s, 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 simply 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 a lot of binary operations have to be performed.

6 Instructions timings

We have fully implemented our proposed instruction set. We summarize the timings in Table 4 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. For the timings, Table 4 first provides estimated values (4th column) obtained by counting the number of blind rotations and key switches (by far the most costly unitary operations) in the *most optimized* versions of each of our instructions multiplied by the measured unitary times of the formers. Then, the last column of that table provides measured execution times. However, only execution times marked with a † are obtained for these *most optimized* implementations. The other measurements are obtained for sub-optimal implementations lacking some MVLUTeval optimizations (i.e., some degree of factorization for blind rotations and key switches). Although our estimations appear reasonably accurate, *we are planning to soon release a new version of this table with all measured timings for the fully optimized implementations.*

7 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.

7.1 Testing a few Elementary Algorithms

Bubble sort – Using our MIN and MAX homomorphic operators, we are able to sort an array. Bubble sorting consists of repeatedly comparing consecutive elements in an array and permuting them when they are 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 by means of MAX and MIN computations as done in algorithm 1. Thus, for all comparisons, we have to compute both MIN and MAX and reassign the results accordingly. To give an order of magnitude for the

Instr.	N_{br}	N_{ks}	ms*	ms	Instr.	N_{br}	N_{ks}	ms*	ms
ANDi/ORi/XORi	2	0	60	69†	AND/OR/XOR	4	2	260	278†
DC	2	0	60	81†	RC	8	0	240	267†
SHLi/SHRi	2	0	60	72†	SHL/SHR	6	4	460	545
ROLi/RORi	2	0	60	125	ROL/ROR	9	6	690	819
EQi	2	0	60	88†	EQ	6	3	390	393†
LT(E)i/GT(E)i	2	1	130	126†	LT(E)/GT(E)	9	5	390	671
(N)CDUP	3	1	160	262	CSEL	9	4	550	791
NEG/ABS	2	1	130	215†	MIN/MAX	16	10	1080	1368
ADDi/SUBi	2	1	130	137†	ADD/SUB	7	4	490	493†
ADDZ	4	2	260	271†	MUL(M)i/DIV(4)i/MOD4i	2	1	130	262
MODi	3	2	230	267†	MUL	10	6	720	819
MULM	31	20	2070	3014	DIV4	20	12	1440	2377
DIV	97	56	6830	7711†	MOD4	10	7	790	952
MOD	91	50	6230	8749	(N)CDUPi	1	0	30	63

Table 4: Mnemonics, blind rotations and keyswitches counts as well as execution times for our (T)FHE processor abstraction instruction set. See Section 6, p.16 for further comment.

execution time, sorting an array of five ciphertexts encrypting 8-bit values using this "sorting in place" algorithm takes around 16 seconds. Execution timings can be found in Table 5.

Algorithm 1 Homomorphic bubble sort

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**
    $C_t \leftarrow \mathsf{MIN}(A[j], A[j+1])$
    $A[j+1] \leftarrow \mathsf{MAX}(A[j], A[j+1])$
    $A[j] \leftarrow C_t$
return A

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 5).

Average of array elements – 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 5, when tried on a five-element array, this computation takes less than 4 seconds.

Array dereferencing and assignment – 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. Note that 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 2.3.1), where we rotate a new *encrypted* test polynomial by an encrypted index. Dereferencing arrays with more than 256 elements is also possible but we do not detail that. Lastly,

Algorithm 2 Homomorphic maximum of an array

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 $\bar{C}$

Algorithm 3 Homomorphic average of array elements

Input: A an array of n encryptions of 8-bit values
Output: $\bar{C}$ 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 $\bar{C} = (C_i, C_d)$

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 5, the sequential evaluation

Algorithm 4 Homomorphic array 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

of this operator on an array of five 8-bit encrypted inputs takes 4.45 seconds.

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 "$\mathsf{while}\ c(S)\ \mathsf{do}\ S := f(S)$" can be rewritten as "$\mathsf{for}\ 0 \le i < B\ \mathsf{do}\ \mathsf{if}\ c(S)\ \mathsf{then}\ S := S\ \mathsf{else}\ S := f(S)$". In essence, that latter form computes a fixed point after condition $c(S)$ reaches a true value, and the inner $\mathsf{if\text{-}then\text{-}else}$ statement can then be done via a CSEL instruction.

7.2 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. 5.2) 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. 5.2, 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

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

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:

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

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 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. 4.3 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. By 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 5.

8 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 langage 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. Indeed, our approach

Table 5: Execution times of different homomorphic algorithms.

Algorithm	Execution Time (secs)
Bubble sort	16
Minimum/maximum	4.71
Average	3.68
Array assignment	4.45
Neuron with sigmoid	16
Neuron with Heaviside	6.85

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.

References

[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*, pages 19–33, 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*, pages 91–109, 2023.

[BDGM19] Z. Brakerski, N. Döttling, S. Garg, and G. Malavolta. Leveraging linear decryption: Rate-1 fully-homomorphic encryption and time-lock puzzles. In *TCC*, pages 407–437, 2019.

[BMMP18] F. Bourse, M. Minelli, M. Minihold, and P. Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *CRYPTO*, page 483–512, 2018.

[BPR23] N. Bon, D. Pointcheval, and M. Rivain. Optimized Homomorphic Evaluation of Boolean Functions. Cryptology ePrint Archive, Paper 2023/1589, 2023.

[BPS12] M. Brenner, H. Perl, and M. Smith. How practical is homomorphically encrypted program execution? an implementation and performance evaluation. In *IEEE TrustCom*, pages 375–382, 2012.

[CBSZ23] P-E. Clet, A. Boudguiga, R. Sirdey, and M. Zuber. *ComBo: A Novel Functional Bootstrapping Method for Efficient Evaluation of Nonlinear Functions in the Encrypted Domain*, pages 317–343. 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*, pages 313–333, 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.

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

[CGRS14] D. B. Cousins, J. Golusky, K. Rohloff, and D. Sumorok. An FPGA co-processor implementation of homomorphic encryption. In *IEEE HPEC*, pages 1–6, 2014.

[CIM18] S. Carpov, M. Izabachène, and V. Mollimard. New techniques for Multi-value input Homomorphic Evaluation and Applications. Cryptology ePrint Archive, Paper 2018/622, 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*, pages 1–19, Cham, 2021. Springer International Publishing.

[CKK19] J.H. Cheon, D. Kim, and D. Kim. Efficient homomorphic comparison methods with optimal complexity. Cryptology ePrint Archive, Paper 2019/1234, 2019.

[CLOT21] I. Chillotti, D. Ligier, J-B. Orfila, and S. Tap. Improved Programmable Bootstrapping with Larger Precision and Efficient Arithmetic Circuits for TFHE. Cryptology ePrint Archive, Report 2021/729, 2021.

[CS19] A. Chatterjee and I. Sengupta. FURISC: FHE encrypted URISC design. In *Fully Homomorphic Encryption in Real World Applications*, pages 87–115. 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.

[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*, pages 284–290, 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):229–253, Feb. 2021.

[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*, pages 702–708, 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*, pages 501–537, 11 2022.

[LLKN21] E. Lee, J-W. Lee, Y-Sik. Kim, and J-S. No. Optimization of homomorphic comparison algorithm on RNS-CKKS scheme. Cryptology ePrint Archive, Paper 2021/1215, 2021.

[PJH23] Méaux P., Park J., and V. L. Pereira H. Towards practical transciphering for FHE with setup independent of the plaintext space. Cryptology ePrint Archive, Paper 2023/1531, 2023.

[SV10] N. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *PKC*, page 420–443, 2010.

[TCBS23a] D. Trama, P-E. Clet, A. Boudguiga, and R. Sirdey. *Building Blocks for LSTM Homomorphic Evaluation with TFHE*, pages 117–134. 06 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, page 79–90. Association for Computing Machinery, 2023.

[TM13] N. G. Tsoutsos and M. Maniatakos. Investigating the application of one instruction set computing for encrypted data computation. In *SPACE*, pages 21–37, 2013.

[TM14] N. G. Tsoutsos and M. Maniatakos. HEROIC: Homomorphically encrypted one instruction computer. In *IEEE DATE*, pages 1–6, 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.