Updates & Errata

February 2022

In Strategy I-B: Ternary Gates, we corrected Equation 16: from $(a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge c)$, which is true iff *exactly* two variables are true, to $(a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$, which is true iff *at least* two variables are true. The textual description was correct: "... is true iff at least two of the input variables are true". Also the homomorphic evaluation proposed in (18), which uses the binary TFHE encoding, was correct.

Parallel Operations over TFHE-Encrypted Multi-Digit Integers *

Jakub Klemsa(✉)[1,2] and Melek Önen[2]

[1]CTU in Prague, Prague, Czechia
[2]EURECOM, Sophia-Antipolis, France

jakub.klemsa@fel.cvut.cz, melek.onen@eurecom.fr

Abstract

Recent advances in Fully Homomorphic Encryption (FHE) allow for a practical evaluation of non-trivial functions over encrypted data. In particular, novel approaches for combining ciphertexts broadened the scope of prospective applications. However, for arithmetic circuits, the overall complexity grows with the desired precision and there is only a limited space for parallelization.

In this paper, we put forward several methods for fully parallel addition of multi-digit integers encrypted with the TFHE scheme. Since these methods handle integers in a special representation, we also revisit the signum function, firstly addressed by Bourse et al., and we propose a method for the maximum of two numbers; both with particular respect to parallelization. On top of that, we outline an approach for multiplication by a known integer.

According to our experiments, the fastest approach for parallel addition of 31-bit encrypted integers in an idealized setting with 32 threads is estimated to be more than 6× faster than the fastest sequential approach. Finally, we demonstrate our algorithms on an evaluation of a practical neural network.

Index terms—Fully homomorphic encryption; TFHE scheme; parallelization; parallel addition

1 Introduction

Fully Homomorphic Encryption (FHE) is a technique that allows to evaluate arbitrary function over encrypted data. Since the breakthrough by Gentry [16], FHE is experiencing a remarkable boom. Hand by hand with the development of new schemes, the range of proposed applications is still growing, reaching use cases like cloud-assisted neural network inference [4, 10, 18, 20, 24, 28].

*This paper is to be presented at ACM CODASPY'22, April 24–27, 2022, Baltimore, MD, USA. This is the full version of the paper including appendices. The proceedings version of this paper can be found at https://doi.org/10.1145/3508398.3511527.

Related Work

Since their very beginnings, FHE constructions build upon encryption schemes that add certain amount of noise to the message. With each homomorphic operation, the amount of noise grows, which can—at some level—prevent the correct decryption of the message. The key invention by Gentry to counter this problem is a method referred to as *bootstrapping*, which aims at refreshing the noise to a certain, fixed level. Initially, bootstrapping was a very costly operation [17], however, nowadays, modern schemes achieve convenient execution times on an ordinary hardware, which makes practical applications feasible [4, 10, 18, 20, 24, 28, 5, 9, 19].

Among other somewhat/fully homomorphic schemes like BGV [6], BFV [15], CKKS [8], or FHEW [14], we have chosen the TFHE Scheme by Chillotti et al. [9], which is fully homomorphic, hence versatile, and which achieves the fastest bootstrapping times (under 0.1 s). As a convenient benefit, the TFHE bootstrapping procedure is capable of evaluating a *Look-Up Table* (LUT) at no additional cost; referred to as *Programmable Bootstrapping* (PBS; [10]), or *functional bootstrapping* [19]. However, the size of the TFHE message space (hence that of the evaluated LUT) is fairly limited: the bootstrapping complexity grows roughly exponentially with the guaranteed message bit-length [24].

Problem Statement

In a recent paper [11], authors identify altogether eight limitations of TFHE, among them is the limited message space size (point B in [11]). To overcome this limitation, several authors suggest to split a long input into smaller chunks and then evaluate a series of LUTs: either using the binary message space for arbitrary functions [9], or using a multivalue message space for the comparison of unbounded integers [5], for integer addition [24, 28], or for arbitrary functions [19, 11]. Another limitation of TFHE, which is not fully addressed yet, is the fact that the bootstrapping operation is not multi-thread friendly (point C in [11]; resolved to certain extent).

In this paper, we focus on this particular drawback: we suggest to mitigate its impact by parallel execution of several instances of TFHE bootstrapping on specifically tailored inputs. Namely, we focus on integer addition, for which we propose a method that can process encrypted inputs *completely* in parallel, independent of the input length. Our method is based on a meta-algorithm by Chow et al. [12], which allows to build custom algorithms for parallel addition, and which uses a non-standard integer representation. Although the representation is non-standard, it does not limit us to only addition: indeed, we revisit and outline parallel algorithms for signum, maximum, and scalar multiplication in this representation. We keep this representation across all presented operations and we avoid unnecessary conversions, which cannot be done in parallel. Our methods aim at improving the applicability of arithmetic circuits, which can be used, e.g., for statistics or neural network evaluation. With our approach and sufficient resources, evaluation using high-precision integers becomes no more a limit.

Our Contributions

We put forward several approaches to parallelize various operations over TFHE-encrypted integer inputs. The main idea behind the newly proposed algorithms is to (i) convert the input integers into a special multi-digit representation, (ii) encrypt each digit with TFHE, and (iii) execute dedicated algorithms to evaluate integer operations over the encrypted digits *in parallel*. More specifically:

- We propose two families of algorithms for parallel addition of encrypted integers depending on the underlying representation: the first family operates over a base-2 representation, whereas the second one sets the base to 4.
- For each family, three incremental strategies are outlined: The goal of the first strategy is to choose the smallest message space of the underlying TFHE; The second strategy aims at decreasing the number of bootstraps by allowing slight increase on the TFHE message space; Finally, the third strategy develops an algorithm with the minimum number of bootstrap operations (two in this context).
- We suggest an improvement to the standard addition with carry as implemented in the TFHE Library [27] by introducing ternary logical gates, which decreases the number of bootstraps from five to two.
- We revisit an algorithm for signum by Bourse et al. with respect to the special integer representation, and we employ this method to construct a new algorithm for the maximum of two integers.
- Finally, we outline an approach for the multiplication of an encrypted number by a known integer, which we employ for a demo evaluation of a neural network over encrypted input data.
- A thorough comparison of the proposed and revisited algorithms is provided both in terms of analytical complexity and experimental execution time, with particular respect to parallelization. It follows from our experiments that we can achieve more than $6\times$ faster addition of 31-bit integers (using 32 parallel bootstrapping threads) compared to the standard addition with carry.

Paper Outline

In Section 2, we recall algorithms for addition of multi-digit integers, in particular those parallelizable, and we provide a brief overview of the TFHE scheme and its variants. Next, we propose several approaches for the transformation of algorithms for integer addition, signum and maximum of two numbers into the TFHE-encrypted domain in Section 3. In Section 4, we compare and benchmark the proposed methods and we discuss the results. Finally, we conclude the paper in Section 5.

2 Preliminaries

In this section, we discuss the choice of an algorithm for parallel addition of multi-digit integers. Next, we revisit two variants of the TFHE Scheme: (i) one with the binary message space, and (ii) one with a multivalue message space.

2.1 Algorithms for Multi-Digit Integer Addition

Before we deal with algorithms for multi-digit integer addition, we briefly recall a notation used for integer representation and we introduce two auxiliary functions. Note that we provide a complete overview of notation in Appendix A.

Integer Representation

For a base $\beta \in \mathbb{N}$, $\beta \geq 2$, we call $\mathbf{x} \in \{0, 1, \ldots, \beta - 1\}^n$, $\mathbf{x} = (x_{n-1} \ldots x_1 x_0 \bullet)_\beta$, the *standard base-$\beta$ representation of $X \in \mathbb{N}$* iff

$$X = \sum_{i=0}^{n-1} \beta^i x_i. \tag{1}$$

For i out of the range $[0, n)$, we assume $x_i = 0$.

Threshold Functions

Let $b \in \mathbb{N}$. We introduce the following functions:

$$f_b(x) = \begin{cases} -1 & \ldots\ x \leq -b, \\ 0 & \ldots\ -b < x < +b, \\ +1 & \ldots\ +b \leq x, \end{cases} \tag{2}$$

$$g_b(x) = \begin{cases} -1 & \ldots\ x = -b, \\ 0 & \ldots\ x \neq \pm b, \\ +1 & \ldots\ x = +b. \end{cases} \tag{3}$$

We will use the notations $x \gtreqless \pm b$ and $x \equiv \pm b$ for $f_b(x)$ and $g_b(x)$, respectively.

2.1.1 Standard Addition with Carry

In Algorithm I, we recall the standard sequential algorithm for addition with carry, using the standard integer representation with base $\beta \in \mathbb{N}$, $\beta \geq 2$. Recall that $\div$ stands for integer division, i.e., $a \div b = \lfloor a/b \rfloor$, $b \neq 0$.

Algorithm I Addition with carry using the standard integer representation.

Input: base-β representations $\mathbf{x}, \mathbf{y} \in \{0, \ldots, \beta - 1\}^n$ of $X, Y \in \mathbb{N}_0$, for some $n \in \mathbb{N}$,
Output: base-β representation $\mathbf{z} \in \{0, \ldots, \beta - 1\}^{n+1}$ of $Z = X + Y$.

1: $c \leftarrow 0$
2: **for** $i = 0, 1, \ldots, n - 1$ **do**
3: $\quad w_i \leftarrow x_i + y_i + c$
4: $\quad z_i \leftarrow w_i \bmod \beta$
5: $\quad c \leftarrow w_i \div \beta$
6: $z_n \leftarrow c$
7: **return z**

2.1.2 Algorithms for Parallel Addition

In 1961, Avizienis [3] presented a parameterizable algorithm for parallel addition of multi-digit integers. Later in 1978, Chow et al. [12] further improved the algorithm so that the algorithm can work with smaller alphabets and even with the minimum base $\beta = 2$.

The algorithms input the integers in a special representation: unlike the standard base-β representation, which uses the alphabet $\mathcal{A}_\beta = \{0, 1, \ldots, \beta - 1\}$, this representation uses an alphabet, which (i) contains negative numbers (represented with bars, e.g., $\bar{2} = -2$), which (ii) is symmetric around zero (e.g., $\bar{\mathcal{A}} = \{\bar{2}, \bar{1}, 0, 1, 2\}$), and which (iii) leads to a redundant number representation. We can illustrate redundancy on two different representations of the integer 2, using base $\beta = 4$ and the aforementioned alphabet $\bar{\mathcal{A}}$, as follows: $(1\bar{2}\bullet)_4 = 1 \cdot 4^1 + (-2) \cdot 4^0 = 0 \cdot 4^1 + 2 \cdot 4^0 = (02\bullet)_4$. Finally in 1999, Kornerup [23] showed that a redundant number representation is a necessary condition for a parallel addition algorithm to exist.

In this paper, we aim at turning chosen variants of the parallel addition algorithm into the TFHE-encrypted domain. We have chosen the bases β equal to a power of two, which yields particularly efficient conversions from and into the standard binary representation, namely:

(a) $\beta = 2$ and $\bar{\mathcal{A}}_2 = \{\bar{1}, 0, 1\}$, and

(b) $\beta = 4$ and $\bar{\mathcal{A}}_4 = \{\bar{2}, \bar{1}, 0, 1, 2\}$.

Find the selected parallel addition algorithms as Algorithms IIa (base $\beta = 2$) and IIb (base $\beta = 4$).

Algorithm IIa Parallel addition with $\beta = 2$ and $\bar{\mathcal{A}}_2 = \{\bar{1}, 0, 1\}$.

Input: $(2, \bar{\mathcal{A}}_2)$-representations $\mathbf{x}, \mathbf{y} \in {\bar{\mathcal{A}}_2}^n$ of $X, Y \in \mathbb{Z}$, for some $n \in \mathbb{N}$,
Output: $(2, \bar{\mathcal{A}}_2)$-representation $\mathbf{z} \in {\bar{\mathcal{A}}_2}^{n+1}$ of $Z = X + Y$.

1: **for** $i \in \{0, 1, \ldots, n\}$ **in parallel do**
2: $\quad w_i \leftarrow x_i + y_i$ ▷ sync threads
3: $\quad q_i \leftarrow w_i \gtrless \pm 2 \vee (w_i \equiv \pm 1 \wedge w_{i-1} \gtrless \pm 1)$ ▷ sync threads
4: $\quad z_i \leftarrow w_i - 2q_i + q_{i-1}$
5: **return z**

Algorithm IIb Parallel addition with $\beta = 4$ and $\bar{\mathcal{A}}_4 = \{\bar{2}, \bar{1}, 0, 1, 2\}$.

Input: $(4, \bar{\mathcal{A}}_4)$-representations $\mathbf{x}, \mathbf{y} \in {\bar{\mathcal{A}}_4}^n$ of $X, Y \in \mathbb{Z}$, for some $n \in \mathbb{N}$,
Output: $(4, \bar{\mathcal{A}}_4)$-representation $\mathbf{z} \in {\bar{\mathcal{A}}_4}^{n+1}$ of $Z = X + Y$.

1: **for** $i \in \{0, 1, \ldots, n\}$ **in parallel do**
2: $\quad w_i \leftarrow x_i + y_i$ ▷ sync threads
3: $\quad q_i \leftarrow w_i \gtrless \pm 3 \vee (w_i \equiv \pm 2 \wedge w_{i-1} \gtrless \pm 2)$ ▷ sync threads
4: $\quad z_i \leftarrow w_i - 4q_i + q_{i-1}$
5: **return z**

Note 1. *In both Algorithms IIa and IIb on line 3, we abuse notation and we combine the functions $x \gtrless \pm b$ and/or $x \equiv \pm b'$ with logical operations. Unless $+1$ meets -1 in such an expression, we treat $+1$'s or -1's as a logical 1's and we keep their positive or negative signs, respectively. In case -1* does *meet $+1$ (e.g., $-1 \wedge +1$), we evaluate the expression as 0.*

E.g., for $w_i = +1$ and $w_{i-1} = -2$ in Algorithm IIa, we have $0 \vee (+1 \wedge -1) = 0 \vee 0 = 0$.

Thread Synchronization In Algorithms IIa and IIb, it is not always necessary to synchronize the threads, as commented on lines 2 and 3. There are two alternative approaches:

1. Each worker i calculates all required intermediate values itself (i.e., w_i, w_{i-1}, w_{i-2}, q_i, and q_{i-1}), then synchronization is not needed. Or,
2. each worker calculates his w_i, waits for others, takes w_{i-1} from his neighbor and calculates q_i, waits for others, takes q_{i-1} from his neighbor and finally he calculates z_i.

The decision, which approach to choose, must be made with respect to the target platform and the cost of each operation, in particular after turning the calculations into the TFHE domain.

Conversion from Binary to Redundant For the conversion from the standard binary representation $\mathbf{b}$ to the redundant $(4, \bar{\mathcal{A}}_4)$-representation, we can apply the following procedure (note that no conversion is needed for the $(2, \bar{\mathcal{A}}_2)$-representation):

1. split $\mathbf{b}$ into blocks b_i of 2 bits,
2. if $b_i = 3$ then $x_i \leftarrow 2$ and $y_i \leftarrow 1$,
   otherwise $x_i \leftarrow b_i$ and $y_i \leftarrow 0$,
3. run Alg. IIb on $\mathbf{x} = (x_n \dots x_1 x_0 \bullet)_4$ and $\mathbf{y} = (y_n \dots y_1 y_0 \bullet)_4$.

It follows that the conversion from the binary to the redundant representation can be done in parallel.

Conversion from Redundant to Binary For the opposite direction, there does not exist a parallel algorithm. Indeed, the existence of such a parallel algorithm would contradict the impossibility of parallel addition with a non-redundant number representation – this poses the only non-parallelizable step.

We outline an approach for this conversion only for the $(4, \bar{\mathcal{A}}_4)$ variant. It may proceed as follows:

1. for each i from 0 do:
2. if $x_i < 0$ then $x_i \leftarrow x_i + 4$ and $x_{i+1} \leftarrow x_{i+1} - 1$.

Clearly, the value of the represented number does not change as well as the resulting alphabet is $\{0, 1, 2, 3\}$, except for the leading (n-th) position: x_n may end up to be equal to -1, which happens for a negative number. In such a case, the rest of the number representation $(x_{n-1} \dots x_0)_4$ is already in the standard complement code. Depending on the desired width of the resulting integer representation, we can keep rolling i until it reaches the width. E.g., for an 8-bit representation, we have $(\bar{2}\bar{2}\bar{2}\bullet)_4 \to (\bar{1}3112\bullet)_4 \sim$ `(int8_t)(0b11'01'01'10)` $= -42$.

2.2 The TFHE Scheme

The TFHE encryption scheme by Chillotti et al. [9] is based on a particular variant of the famous Learning With Errors (LWE) scheme, which was first introduced by Regev [26]. The variant, named TLWE, operates over a *torus* plaintext space: denoted by $\mathbb{T}$, the *torus* refers to $\mathbb{R}/\mathbb{Z}$, i.e., the fractional part of real numbers, or reals modulo 1. In a nutshell, TLWE encrypts a plaintext value $\mu \in \mathbb{T}$ into a ciphertext $\mathbf{c} = (\mathbf{a}, b) \in \mathbb{T}^n \times \mathbb{T}$ (also referred to as the TLWE *sample*) as follows:

(i) draw a uniformly random mask $\mathbf{a} \in \mathbb{T}^n$, (ii) draw an error term $e \in \mathbb{T}$ with a zero-centered Gaussian distribution with the standard deviation $\alpha \in \mathbb{R}_0^+$, and (iii) set b as

$$b = \mathbf{s} \cdot \mathbf{a} + \mu + e, \tag{4}$$

where $\mathbf{s} \in \{0,1\}^n$ is the TLWE secret key. For a detailed description of TFHE, we refer to [9].

The most important feature of TFHE is the so-called *bootstrapping*. The original purpose of bootstrapping is to reduce the magnitude of a noise, which must be present in TLWE ciphertexts for security reasons, and which grows with each homomorphic addition operation. As a convenient byproduct, bootstrapping is inherently capable of evaluation of a *negacyclic* function, i.e., a function, for which it holds $f(x + 1/2) = -f(x)$, $x \in \mathbb{T}$. Using certain message representations in the torus, bootstrapping can be either used for the homomorphic evaluation[1] of logical gates [9], or for the evaluation of multivalue Look-Up Tables (LUTs) and/or their compositions [4, 5, 7, 19]. In the following sections, we refer to these variants as the *Binary* TFHE and the *Multivalue* TFHE, respectively.

2.3 Binary TFHE

Chillotti et al. [9] proposed to map the logical values 0 and 1 into the torus plaintext space $[-1/2, +1/2)$ as $-1/8$ and $+1/8$, respectively. Such torus representations are then encrypted with the underlying TLWE. Next, these samples enter various logical gates, which apply homomorphic addition and bootstrapping operations, yielding an encryption of the original $\pm 1/8$ representation (with a small amount of noise).

2.3.1 Binary Logical Gates

Using the aforementioned representation together with homomorphic addition and bootstrapping, in [9], authors suggest to evaluate basic binary logical gates over bits encrypted as TLWE samples $\mathbf{c}$ and $\mathbf{d}$, $\mathbf{c}, \mathbf{d} \in \mathbb{T}^n \times \mathbb{T}$, as follows (we are listing only selected gates):

$$\neg\,\mathbf{c} \quad (\mathbf{NOT}): \quad -\mathbf{c} \quad \text{(requires no bootstrapping)}, \tag{5}$$

$$\mathbf{c} \wedge \mathbf{d} \quad (\mathbf{AND}): \quad \mathbf{c} + \mathbf{d} - 1/8 \gtrless 0, \tag{6}$$

$$\mathbf{c} \vee \mathbf{d} \quad (\mathbf{OR}): \quad \mathbf{c} + \mathbf{d} + 1/8 \gtrless 0, \quad \text{and} \tag{7}$$

$$\mathbf{c} \veebar \mathbf{d} \quad (\mathbf{XOR}): \quad 2(\mathbf{c} + \mathbf{d}) + 1/4 \gtrless 0. \tag{8}$$

For the "$\gtrless$" comparison with 0, they evaluate the following (negacyclic) bootstrapping function:

$$f(x) = \begin{cases} -1/8 & \dots\ x \in [-1/2, 0), \\ +1/8 & \dots\ x \in [0, +1/2). \end{cases} \tag{9}$$

With a pen and paper, it is easy to verify that the gates indeed yield the desired result in the original $\pm 1/8$ encoding.

[1] N.b., this feature makes TFHE fully homomorphic.

2.3.2 The MUX Gate

In addition to the binary logical gates, TFHE implements the MUX *gate*, defined as follows:

$$\mathsf{MUX}(a, b, c) = a \ ? \ b \ : \ c, \tag{10}$$

i.e., the standard ternary expression. The MUX gate is implemented using a translation into the standard binary logical gates as

$$\mathsf{MUX}(a, b, c) = (a \wedge b) \veebar (\neg a \wedge c), \tag{11}$$

hence it requires three bootstraps.

Note 2. *The values* $-1/8$ *and* $+1/8$*, which represent logical* 0 *and* 1 *in the torus, respectively, are* $1/4$ *apart from each other, i.e., we have an equivalent of* 4 *values uniformly spread across the torus. This mandates certain bounds on the noise magnitude to guarantee correct decryption and we will need this later for the comparison with the multivalue* TFHE*.*

In addition, note that during the homomorphic evaluation of the logical gates (except for the **NOT** *gate), a homomorphic addition of two values is performed before bootstrapping. This implies other requirements on the amount of noise in the freshly encrypted, or bootstrapped samples. Interestingly, the* **XOR** *gate only scales* everything *by a factor of* 2*.*

2.4 Multivalue TFHE

In their original paper [9], Chillotti et al. outlined an approach for the evaluation of multivalue LUTs. In the core, the method is based on the evaluation of logical gates using the binary TFHE, however, they introduce interesting packing techniques, which aim at reducing the overall complexity.

For most algorithms presented in this paper, we will employ the *multivalue variant* of TFHE and related LUT evaluation techniques. As opposed to the binary TFHE, which encrypts $\pm 1/8$, in the multivalue TFHE, we split the torus into 2^π equal parts, where π stands for the desired bit-precision of the message. We will refer to the torus values $\{0/2^\pi, 1/2^\pi, \ldots, (2^\pi-1)/2^\pi\}$ directly as the plaintext space of the multivalue TFHE to avoid confusion in subsequent algorithms. Each of these values will represent a number from $\mathbb{Z}_{2^\pi}$.

2.4.1 Notation for Custom Functions

Let us introduce a notation for custom functions represented by a list of function values. Let $f\colon \mathbb{Z}_{2^\pi} \to \mathbb{Z}_{2^\pi}$ be a negacyclic function and let $\mathbf{f} \in \mathbb{Z}_{2^\pi}^{2^{\pi-1}}$ be the list of function values on $[0, 2^{\pi-1})$, the rest is given by its negacyclicity. Then for $x \in \mathbb{Z}_{2^\pi}$, we denote $f(x)$ as $\mathbf{f}[x]$. In addition, for unused function values (i.e., outside of the domain of f), we use the symbol $\circ$, which can be set to, e.g., zeros. Finally, in case we only know a lower bound on π, we use the symbol $\|$ to denote the place to be filled with an appropriate number of $\circ$'s (if applicable), *including the negacyclic extension.* E.g., for $\pi = 3$, the vector $(1, 2 \,\|{-3})$ represents the negacyclic function given by $(1, 2, \circ, 3, -1, -2, \circ, -3)$.

2.4.2 Noise Growth

As already outlined, in the TLWE encryption scheme, certain amount of noise (error) must be added to the message; cf. (4). The error term is additive with respect to the homomorphic addition

operation, while it must not exceed certain bound to decrypt correctly. Based on the expected noise growth, the TFHE parameters must be tuned accordingly to prevent message damage. We will consider the noise growth with respect to error variance.

Let us assume that each fresh(ly bootstrapped) sample has a constant amount of noise in terms of error variance. Then, since error variance is additive with quadratic weights, we can express the error growth after homomorphic addition as the sum of squared weights. We refer to this quantity as the *quadratic weight* and we denote its logarithm by 2Δ. E.g., for $1 \cdot x - 3 \cdot y + 2 \cdot z$, we have the quadratic weight equal to $1^2 + 3^2 + 2^2 = 14$ and $2\Delta = \log_2(14)$. For the binary TFHE, we have $2\Delta = \log_2(2) = 1$; cf. Note 2. Note that Δ itself is intended to express the bit-length of standard deviation.

2.4.3 Evaluating Multi-Word Input LUTs

Recently, Guimarães et al. [19] proposed to distinguish two approaches for the homomorphic evaluation of multi-word input LUTs: (i) a *tree-based method*, and (ii) a *chaining method*. The tree-based method is based on a method referred to as the TLWE-*to*-TRLWE *key switching* and it aggregates intermediate outputs in the next-level LUTs, which makes it suitable for the evaluation of *arbitrary* functions. On the other hand, the chaining method has a fixed set of LUTs organized in a chain, while it linearly-combines intermediate outputs with the next level inputs, hence it is more functionally restricted. Compared to the tree-based method, the chaining method has smaller error growth and it seems particularly suitable for functions with simple, carry-like logic (e.g., standard addition with carry). In this paper, we will be using the chaining method.

3 Parallel Operations over Encrypted Multi-Digit Integers

In this section, we first revisit the standard addition with carry, for which we propose an improved binary variant. The core of this section is the design of algorithms for parallel addition of TFHE-encrypted multi-digit integers. On top of the addition algorithms, we outline a method for multiplication of an encrypted integer by a known integer, referred to as scalar multiplication. We also comment on other operations, how they may proceed in the non-standard number representation that is used for parallel addition, with particular respect to their parallelization. Namely, we select the signum function and the maximum of two numbers, which can be employed for a neural network evaluation. In contrast to many other works, we provide our algorithms in the *cleartext* domain, which simplifies their reading and understanding. For each algorithm, we provide a complexity analysis: we summarize the requirements on the (multivalue) TFHE parameters (plaintext space, quadratic weights) and we evaluate the number of bootstraps, the ideal number of threads, and the circuit depth.

The Big Picture

Before going into the details, we outline the *big picture* of prospective application of the parallelized operations in the encrypted domain and we provide some technical observations. In five steps, we summarize the data flow from the user (U), through the cloud (C), and back to the user:

(U) convert sensitive data into selected redundant integer representation (e.g., $(4, \bar{\mathcal{A}}_4)$),

(U) encrypt converted data digit-by-digit with TFHE using appropriate parameters; send it for processing to the cloud,

(C) evaluate function f over the encrypted data; send the (still encrypted) result back to the user,

(U) decrypt result (still in the redundant representation),

(U) convert result back to the native representation.

See Figure 1 for a detailed overview of the whole life-cycle.

Input: $\mathbf{X} \in \mathbb{Z}^M$: $\mathbf{X} = (X_i)_{i=1}^M$, $M \in \mathbb{N}$

Input: $f\colon \mathbb{Z}^M \to \mathbb{Z}^N$, $N \in \mathbb{N}$, f decomposed to additions/subtractions/signums/maximums

$\mathbf{x}_i = \left(x^{(i)}_{m_i-1} \dots x^{(i)}_1 x^{(i)}_0 \bullet\right)_2$	binary representation of X_i (plain)	
$\downarrow$	**(P)** *convert from binary to* $(4, \bar{\mathcal{A}}_4)$	user
$\mathbf{x}'_i = \left(x'^{(i)}_{m'_i-1} \dots x'^{(i)}_1 x'^{(i)}_0 \bullet\right)_4$	$(4, \bar{\mathcal{A}}_4)$-representation of X_i (plain)	
$\downarrow$	**(P)** *encrypt digits into* $\mathbf{c}^{(i)}_{j_i} = \mathsf{TFHE.Encr}\left(x'^{(i)}_{j_i}\right)$	
$\mathbf{C}_i = \left(\mathbf{c}^{(i)}_{m'_i-1}, \dots, \mathbf{c}^{(i)}_1, \mathbf{c}^{(i)}_0\right)$	$(4, \bar{\mathcal{A}}_4)$-representation of X_i (encrypted)	$\downarrow$
$\downarrow$	**(P*)** *homomorphically evaluate decomposed* f	cloud
$\mathbf{D}_k = \left(\mathbf{d}^{(k)}_{n'_k-1}, \dots, \mathbf{d}^{(k)}_1, \mathbf{d}^{(k)}_0\right)$	$(4, \bar{\mathcal{A}}_4)$-representation of Y_k (encrypted)	$\downarrow$
$\downarrow$	**(P)** *decrypt digits into* $y'^{(k)}_{l_k} = \mathsf{TFHE.Decr}\left(\mathbf{d}^{(k)}_{l_k}\right)$	
$\mathbf{y}'_k = \left(y'^{(k)}_{n'_k-1} \dots y'^{(k)}_1 y'^{(k)}_0 \bullet\right)_4$	$(4, \bar{\mathcal{A}}_4)$-representation of Y_k (plain)	user
$\downarrow$	**(S)** *convert from* $(4, \bar{\mathcal{A}}_4)$ to binary	
$\mathbf{y}_k = \left(y^{(k)}_{n_k-1} \dots y^{(k)}_1 y^{(k)}_0 \bullet\right)_2$	binary representation of Y_k (plain)	

Output: $\mathbf{Y} = f(\mathbf{X}) \in \mathbb{Z}^N$: $\mathbf{Y} = (Y_k)_{k=1}^N$, $N \in \mathbb{N}$

Figure 1: Flow of data $\mathbf{X}$: from the user, where $\mathbf{X}$ gets encrypted, through the server, where a function f is homomorphically evaluated over the encrypted data, back to the user, where the result is decrypted and it yields the desired $\mathbf{Y} = f(\mathbf{X})$. For each step, we stress whether: **(P)** it can be done in parallel in a constant number of steps, or **(P*)** it may involve parallel reduction, which requires $O(\log n)$ steps (whenever sgn or max is evaluated), or **(S)** it can only be computed sequentially.

Minimizing the Plaintext Space

As previously mentioned, the TFHE bootstrapping function is inherently negacyclic, therefore we need to deal with this limitation, in case we need to evaluate a function that is not negacyclic. In [7, 19], authors suggest to employ only one half of the plaintext space, which is the generic approach. On the other hand, Klemsa et al. [22] suggest to exploit any possible overlap of the negacyclic extension (cf. Figure 3 in their paper), which may lead to plaintext space savings. Recall

that the complexity of the multivalue TFHE grows roughly exponentially with the plaintext precision π [24], therefore our aim is to use as little π as possible, given the function. Let us discuss the functions $x \gtrless \pm b$, $x \equiv \pm b$ (as introduced in (2) and (3)) and a shifting trick.

Observation 1. *For the negacyclic extension of both $x \gtrless \pm b$ and $x \equiv \pm b$ on the desired domain $[-a, +a]$, where $a \geq b > 0$, we choose the range $[-a - b, a + b)$, which is the minimal range for $x \gtrless \pm b$, and which is sufficient for $x \equiv \pm b$. We set the negacyclic extension function f, $f\colon [-a-b, a+b) \to \{-1, 0, 1\}$, on the non-negative half of the domain as follows:*

$$f(x) = \begin{cases} 0 & x \in [0, b-1], \\ 1 & x \in [b, a], \\ 0 & x \in [a+1, a+b-1]. \end{cases} \tag{12}$$

It can be easily verified that the negacyclic extension of f includes the function $x \gtrless \pm b$ on the domain $[-a, +a]$; cf. Appendix B.

Observation 2. *Thanks to the cheap additive homomorphism, one may also shift the function by a constant, if this helps to find the negacyclic extension on a smaller domain. E.g., the function $f\colon \mathbb{Z}_4 \to \mathbb{Z}_4$, $f(0) = 1$, $f(1) = 2$, $f(2) = 1$, $f(3) = 0$, is not negacyclic, but $f(x) - 1$ is. Therefore, it is not needed to extend the domain to $\mathbb{Z}_8$—it is sufficient to use the negacyclic $f(x) - 1$ over $\mathbb{Z}_4$ and add $+1$ to the result.*

Moreover, it is possible to apply the shifts by $1/2$, which allows to evaluate other functions as well. E.g., $f\colon \mathbb{Z}_4 \to \mathbb{Z}_4$, $f(0) = 0$, $f(1) = 0$, $f(2) = 1$, $f(3) = 1$ can be evaluated using the shift by $-1/2$. N.b., this trick is in principle used by Chillotti et al. for their construction of binary logical gates.

3.1 Addition with Carry Revisited

First, let us revisit the standard addition with carry. We focus on the evaluation of z_i and c in the main for-loop in Algorithm I, given x_i, y_i and c from the previous round (w_i is only an auxiliary variable, which does not need to be explicitly calculated). In the TFHE Library [27], there are actually two approaches implemented, however, both work with base $\beta = 2$ and both need 5 bootstraps of the binary TFHE, hence we cover both as Strategy I-A. Next, we propose an improvement to this strategy, which employs our ternary gates and which requires only 2 bootstraps per bit, referred to as Strategy I-B. Finally, we revisit the approach by Guimarães et al., who use base $\beta = 4$ and the multivalue TFHE with $\pi = 3$, referred to as Strategy I-C.

3.1.1 Strategy I-A: Original TFHE Library

In the TFHE Library, $z_i \leftarrow x_i \veebar y_i \veebar c$ is evaluated using two XOR gate bootstraps. Next, two approaches to update the carry c are implemented: either

1. without the MUX gate as
$$c = (x_i \cdot y_i) + (c \cdot w_i), \quad \text{or} \tag{13}$$
2. with the MUX gate as
$$c = w_i \ ?\ c : (x_i \cdot y_i). \tag{14}$$

Either way requires 5 bootstraps per digit (1 bit) in total. Find the variant without the MUX gate in Algorithm I-A, where we additionally propose a simple parallelization.

Algorithm I-A Addition with carry, $\beta = 2$ and 5 bootstraps using binary TFHE (variant without MUX; [27]).

Input: one bit of each addend x_i and y_i, previous carry c_{i-1},
Output: one bit of result z_i, new carry c_i.

1: **in parallel do** ▷ our suggestion
2: $t \leftarrow x_i \veebar y_i$
3: $u \leftarrow x_i \wedge y_i$
4: **in parallel do** ▷ our suggestion
5: $z_i \leftarrow t \veebar c_{i-1}$
6: $v \leftarrow t \wedge c_{i-1}$
7: $c_i \leftarrow u \veebar v$
8: **return** (z_i, c_i)

Analysis As outlined in Note 2, for the binary TFHE, we need an equivalent of $\pi = 2$ and $2^{2\Delta} = 2$ (see Section 2.4.2 for the parameter 2Δ). We propose to calculate the pairs (t, u) and (z_i, v) in parallel, hence we make use of 2 threads and they require 3 steps per digit.

3.1.2 Strategy I-B: Ternary Gates

Let us introduce the following ternary logical gates:

$$\mathbf{XOR3}(a, b, c) \coloneqq a \veebar b \veebar c, \quad \text{and} \tag{15}$$

$$\mathbf{2OF3}(a, b, c) \coloneqq (a \wedge b) \vee (a \wedge c) \vee (b \wedge c), \tag{16}$$

where the latter is true iff at least two of the input variables are true. These gates can be evaluated using the binary TFHE encoding (i.e., $\pm 1/8$) in a single bootstrap as follows:

$$\mathbf{XOR3}(\mathbf{a}, \mathbf{b}, \mathbf{c}) : \quad -2(\mathbf{a} + \mathbf{b} + \mathbf{c}) \gtrless 0, \quad \text{and} \tag{17}$$

$$\mathbf{2OF3}(\mathbf{a}, \mathbf{b}, \mathbf{c}) : \quad (\mathbf{a} + \mathbf{b} + \mathbf{c}) \gtrless 0, \tag{18}$$

cf. (9) for the "$\gtrless$" comparison with zero in the binary TFHE. With these gates, 2 bootstraps per digit (1 bit) are sufficient; see Algorithm I-B.

Algorithm I-B Addition with carry, $\beta = 2$ and 2 bootstraps using binary TFHE.

Input: one bit of each addend x_i and y_i, previous carry c_{i-1},
Output: one bit of result z_i, new carry c_i.

1: **in parallel do**
2: $z_i \leftarrow XOR3(x_i, y_i, c_{i-1})$
3: $c_i \leftarrow 2OF3(x_i, y_i, c_{i-1})$
4: **return** (z_i, c_i)

Analysis Like in the previous strategy, we have an equivalent of $\pi = 2$ for the binary TFHE. However, unlike *binary* logical gates of the binary TFHE, our *ternary* logical gates require *three*

samples to be added before bootstrapping, which yields the quadratic weight $2^{2\Delta} = 3$. Next, we suggest to calculate (z_i, c_i) in parallel, making use of 2 threads, which only run one bootstrapping (per digit).

3.1.3 Strategy I-C: Switching to Base 4

Guimarães et al. [19] suggest an addition algorithm that uses base $\beta = 4$ and the multivalue TFHE with $\pi = 3$. For the comparison with other methods, we rewrite (and fix) this method into the cleartext domain, as well as we suggest to bootstrap the result one more time, which is not performed in the original method.

Algorithm I-C Addition with carry, $\beta = 4$ and 2 bootstraps using multivalue TFHE with $\pi = 3$ ([19]; corrected, modified).

Input: 2-bit digit of each addend x_i and y_i,
Input: previous result z_{i-1}, previous carry c_{i-1},
Output: 2-bit digit of result z_i, new carry c_i.

1: $w_i \leftarrow x_i + y_i + c_{i-1}$
2: **in parallel do**
3: $\quad z_{i-1} \leftarrow z_{i-1}$ ▷ bootstrap with identity
4: $\quad c_i \leftarrow \left(-\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right)[w_i] + \frac{1}{2}$
5: $z_i \leftarrow w_i - 4 \cdot c_i$
6: **return** (z_i, c_i)

Remarks On line 4, Observation 2 is applied. I.e., c_i evaluates as 0 or 1 if $w_i < 4$ or $w_i \geq 4$, respectively.

Analysis Due to line 5, we have $2^{2\Delta} = 19$. Next, since we bootstrap in parallel, we make use of 2 threads and we only need one bootstrapping step per digit.

3.2 Parallel Addition

Next, we discuss the two chosen variants of parallel addition algorithm as per Algorithms IIa ($\beta = 2$) and IIb ($\beta = 4$). Using the multivalue TFHE with a sufficiently large plaintext space, we suggest to evaluate w_i and z_i via additive homomorphism. For the evaluation of q_i, which is non-linear, and which is illustrated in Figures 2 and 3, respectively, we propose the following three strategies for each variant of the parallel addition algorithm:

1. using the smallest plaintext space possible ($\pi_2 = 3$, $\pi_4 = 4$) and 5 bootstraps as Strategy IIa-D and IIb-G, respectively,
2. using larger plaintext space ($\pi_2 = 4$, $\pi_4 = 5$) and 4 bootstraps as Strategy IIa-E and IIb-H, respectively, and
3. using $\pi_2 = 5$, $\pi_4 = 7$ and only 2 bootstraps as Strategy IIa-F and IIb-I, respectively.

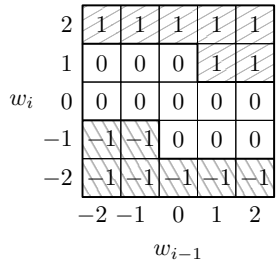

Figure 2: Value of $q_i = q_i(w_{i-1}, w_i)$ as per Algorithm IIa.

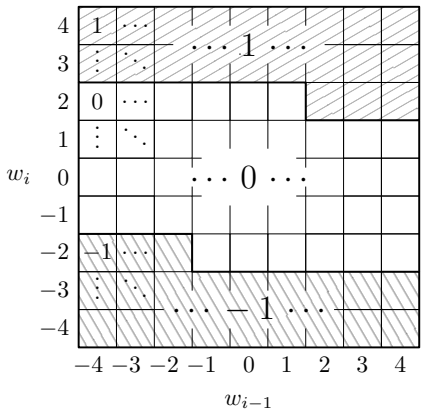

Figure 3: Value of $q_i = q_i(w_{i-1}, w_i)$ as per Algorithm IIb.

Bootstrapping Strategy

We propose the following bootstrapping strategy:

1. from each addition algorithm, return a freshly bootstrapped sample,
2. otherwise, use as little bootstraps as possible,

Ad (1): even with only additions, which do not require bootstrap implicitly, we need to bootstrap time to time anyways; this way, we better control the quadratic weight. Ad (2): in particular, we will not bootstrap after additions, i.e., we will only use bootstrapping to evaluate functions like $x \gtreqless \pm b$ etc.

We calculate required quadratic weight *after* we construct a complete algorithm. N.b., our bootstrapping strategy does not guarantee the best trade-off between the number of bootstraps and the quadratic weight, which affects the TFHE parameters, which in turn affect the time needed for bootstrapping.

3.2.1 Strategy IIa-D and IIb-G: Smallest Plaintext Space

In the first pair of strategies, we aim at using the smallest plaintext space that is needed for the representation of $w_i = x_i + y_i$. For $\beta = 2$ and for $\beta = 4$, we have $|\bar{\mathcal{A}}_2 + \bar{\mathcal{A}}_2| = |\{\bar{2}, \bar{1}, \ldots, 2\}| = 5$ and

$|\bar{\mathcal{A}}_4 + \bar{\mathcal{A}}_4| = |\{\bar{4}, \bar{3}, \ldots, 4\}| = 9$, respectively. It follows that $\pi_2 \geq 3$ and $\pi_4 \geq 4$ is necessary solely for the representation of w_i. We outline the strategies for $\beta = 2$ and $\beta = 4$ in Algorithms IIa-D and IIb-G, respectively.

Algorithm IIa-D Parallel addition with $\beta = 2$, $\pi_2 = 3$ and 5 bootstraps.

Input: $(w_{i-1}, w_i) \in \{\bar{2}, \bar{1}, \ldots, 2\}^2$,
Output: $q_i = w_i \equiv \pm 2 \vee (w_i \equiv \pm 1 \wedge w_{i-1} \gtrless \pm 1)$.

1: **in parallel do**
2: $\quad r_1 \leftarrow w_i \gtrless \pm 2$
3: $\quad r_2 \leftarrow w_i \equiv \pm 1$
4: $\quad r_3 \leftarrow w_{i-1} \gtrless \pm 1$
5: $r_{23} \leftarrow r_2 + r_3 \equiv \pm 2$
6: **return** $q_i \leftarrow r_1 + r_{23}$

Algorithm IIb-G Parallel addition with $\beta = 4$, $\pi_4 = 4$ and 5 bootstraps.

Input: $(w_{i-1}, w_i) \in \{\bar{4}, \bar{3}, \ldots, 4\}^2$,
Output: $q_i = w_i \gtrless \pm 3 \vee (w_i \equiv \pm 2 \wedge w_{i-1} \gtrless \pm 2)$.

1: **in parallel do**
2: $\quad r_1 \leftarrow w_i \gtrless \pm 3$
3: $\quad r_2 \leftarrow w_i \equiv \pm 2$
4: $\quad r_3 \leftarrow w_{i-1} \gtrless \pm 2$
5: $r_{23} \leftarrow r_2 + r_3 \equiv \pm 2$
6: **return** $q_i \leftarrow r_1 + r_{23}$

Analysis In terms of Observation 1, we have $a = 2$ and $b = 2$ due to both lines 2 and 5 of Algorithm IIa-D, hence we need at least the range $[-4, +4)$ to cover the cleartexts in Strategy IIa-D. Note that other lines of Algorithm IIa-D can be covered by this cleartext range, too. Similarly for Strategy IIb-G, we have $a = 4$ and $b = 3$ due to line 2 of Algorithm IIb-G, i.e., Observation 1 mandates the cleartext range $[-7, +7)$. Hence these ranges can be covered using the plaintext space with $\pi_2 = 3$ and $\pi_4 = 4$, respectively, i.e., the originally estimated bounds are sufficient.

In terms of freshly bootstrapped samples, the result writes as $z_i = x_i + y_i - \beta(r_1^{(i)} + r_{23}^{(i)}) + r_1^{(i-1)} + r_{23}^{(i-1)}$, which yields the quadratic weights $2^{2\Delta_2} = 12$ and $2^{2\Delta_4} = 36$. In accordance with our bootstrapping strategy, we apply one more bootstrap to the resulting z_i with the identity mapping, hence in total, we have $4+1$ bootstraps. Note that for both $\beta = 2$ and $\beta = 4$, the identity mapping on $\bar{\mathcal{A}}_\beta$ fits within respective π_β. Both algorithms make use of 3 threads (per digit of $\mathbf{z}$), while running in $2 + 1$ bootstrapping steps (one is for the final bootstrap).

3.2.2 Strategy IIa-E and IIb-H: Save One Bootstrap

In this pair of strategies, we reduce the number of bootstraps by one, while we keep the plaintext space reasonably small, see Algorithms IIa-E and IIb-H for $\beta = 2$ and $\beta = 4$, respectively. As opposed to previous algorithms (e.g., Algorithm IIa-D), we calculate the intermediate value r_{23} in two bootstraps instead of three.

Algorithm IIa-E Parallel addition with $\beta = 2$, $\pi_2 = 4$ and 4 bootstraps.

Input: $(w_{i-1}, w_i) \in \{\bar{2}, \bar{1}, \ldots, 2\}^2$,
Output: $q_i = w_i \equiv \pm 2 \vee (w_i \equiv \pm 1 \wedge w_{i-1} \gtrless \pm 1)$.

1: **in parallel do**
2: $\quad r_1 \leftarrow w_i \gtrless \pm 2$
3: $\quad r_2 \leftarrow 2 \cdot (w_i \equiv \pm 1)$ ▷ scale f_1 by 2
4: $r_{23} \leftarrow w_{i-1} + r_2 \gtrless \pm 3$
5: **return** $q_i \leftarrow r_1 + r_{23}$

Algorithm IIb-H Parallel addition with $\beta = 4$, $\pi_4 = 5$ and 4 bootstraps.

Input: $(w_{i-1}, w_i) \in \{\bar{4}, \bar{3}, \ldots, 4\}^2$,
Output: $q_i = w_i \gtrless \pm 3 \vee (w_i \equiv \pm 2 \wedge w_{i-1} \gtrless \pm 2)$.

1: **in parallel do**
2: $\quad r_1 \leftarrow w_i \gtrless \pm 3$
3: $\quad r_2 \leftarrow 3 \cdot (w_i \equiv \pm 2)$ ▷ scale f_2 by 3
4: $r_{23} \leftarrow w_{i-1} + r_2 \gtrless \pm 5$
5: **return** $q_i \leftarrow r_1 + r_{23}$

Remarks We comment on line 3 of both Algorithms IIa-E and IIb-H that the bootstrapping functions f_1 and f_2 (as per (2)) shall be scaled by 2 and 3, respectively. I.e., we bootstrap directly with the values $\{-2, 0, +2\}$ and $\{-3, 0, +3\}$, respectively. This prevents noise growth that would occur in case of scalar multiplication of the resulting sample.

Analysis By Observation 1 and due to line 4 of both Algorithms IIa-E and IIb-H, we need the ranges $[-7, +7)$ and $[-12, +12)$, respectively, which fit within $\pi_2 = 4$ and $\pi_4 = 5$. The quadratic weights remain equal to 12 and 36, respectively, exactly as in the previous strategies. Compared to the previous strategies, we save one bootstrap – we have $3 + 1$ bootstraps. Finally, each algorithm employs 2 threads (per digit) and runs in $2 + 1$ bootstrapping steps.

3.2.3 Strategy IIa-F and IIb-I: q_i in Single Bootstrap

In this pair of strategies, we evaluate q_i in a single bootstrap as a rounded linear function. Indeed, q_i can be expressed for $\beta = 2$ and $\beta = 4$ as

$$q_i^{(2)} = \left\lfloor \frac{1}{7}\left(w_{i-1}^{(2)} + 3w_i^{(2)}\right) \right\rceil, \quad \text{and} \tag{19}$$

$$q_i^{(4)} = \left\lfloor \frac{1}{27}\left(w_{i-1}^{(4)} + 6w_i^{(4)}\right) \right\rceil, \tag{20}$$

respectively; cf. Figures 2 and 3. With these coefficients, the minimum distance from $\pm 1/2$ is the largest possible. E.g., for $w_{i-1}^{(4)} = 1$ and $w_i^{(4)} = 2$, we round $13/27$ to zero, which is by $1/54$ from $1/2$. Find this approach in Algorithms IIa-F and IIb-I, respectively.

Note 3. *How did we derive* (19) *and* (20)*? We noticed that the values of $q_i = q_i(w_{i-1}, w_i)$ form slant stripes of $-1/0/+1$; cf. Figures 2 and 3. These values can then be obviously expressed as a*

rounded linear function with the slope given by the normal vectors $(1,3)$ *and* $(1,6)$*, respectively, we only need to tune the leading coefficient. We do that by evaluating two neighboring corner values: e.g., for* $\beta = 2$*, we have the corner values at* $(w_{i-1}, w_i) = (0,1)$*, which gives* $w_{i-1} + 3w_i = 3$*, and at* $(1,1)$*, which gives* $w_{i-1} + 3w_i = 4$*, respectively. Hence by setting the coefficient to* $1/7$ *and rounding to integers, we obtain the desired* 0 *and* 1 *at the positions* $(0,1)$ *and* $(1,1)$*, respectively. Similarly, we obtain the coefficient* $1/27$ *for* $\beta = 4$*.*

Algorithm IIa-F Parallel addition with $\beta = 2$, $\pi_2 = 5$ and 2 bootstraps.

Input: $(w_{i-1}, w_i) \in \{\bar{2}, \bar{1}, \ldots, 2\}^2$,
Output: $q_i = w_i \equiv \pm 2 \vee (w_i \equiv \pm 1 \wedge w_{i-1} \gtrless \pm 1)$.

1: **return** $q_i \leftarrow w_{i-1} + 3w_i \gtrless \pm 4$

Algorithm IIb-I Parallel addition with $\beta = 4$, $\pi_4 = 7$ and 2 bootstraps.

Input: $(w_{i-1}, w_i) \in \{\bar{4}, \bar{3}, \ldots, 4\}^2$,
Output: $q_i = w_i \gtrless \pm 3 \vee (w_i \equiv \pm 2 \wedge w_{i-1} \gtrless \pm 2)$.

1: **return** $q_i \leftarrow w_{i-1} + 6w_i \gtrless \pm 14$

Analysis By Observation 1, we need the ranges $[-12, +12)$ and $[-42, +42)$, respectively, which can be covered by $\pi_2 = 5$ and $\pi_4 = 7$. Next, we have the quadratic weights equal to 20 and 74, respectively, due to the calculation of the expression on line 1 of each algorithm. Indeed, the expression rewrites in terms of freshly bootstrapped samples as $x_{i-1} + y_{i-1} + 3/2\beta \cdot (x_i + y_i)$. As intended, the number of bootstraps drops down to $1 + 1$ and each algorithm employs single thread (per digit).

3.3 Scalar Multiplication

By *scalar multiplication* we understand (homomorphic) multiplication of an *encrypted* integer X by a *known* integer k:

$$k \odot \mathsf{TFHE.Encr}(X) \approx \mathsf{TFHE.Encr}(k \cdot X). \tag{21}$$

Note that homomorphic addition is sufficient for this operation – we may employ the schoolbook *double-and-add* approach.

On top of the standard (and naïve) double-and-add, we suggest a two-step optimization:

1. in the binary representation of k, replace any series of 1's (of length 2 and more; take the longest possible) with 0's preceded by 1 and concluded by $\bar{1}$ (e.g., $\ldots 01110 \ldots \rightarrow \ldots 100\bar{1}0 \ldots$; n.b., the value of k remains unchanged) – repeat this step until there are no neighboring 1's,
2. replace any occurrence of $\bar{1}1$ with $0\bar{1}$.

Further optimizations are possible (e.g., exploit repeated patterns), however, this is out of the scope of this paper. To calculate the result, we shift (and possibly negate) $\mathbf{x}$ accordingly to the optimized representation of k, then we use selected addition algorithm to aggregate the shifted (and negated) values. Find our method in Algorithm III, which is also given in the cleartext domain.

Note that for $\beta = 2$, shifting $\mathbf{x}$ by b bits is equivalent to appending/prepending (depends on endianity) a series of b trivial samples of 0 to $\mathbf{x}$. For $\beta = 4$, we suggest to pre-compute double of $\mathbf{x}$ as $\mathbf{y} = \mathbf{x} \oplus \mathbf{x}$, then we append/prepend $\lfloor b/2 \rfloor$ trivial samples of 0 to $\mathbf{x}$ or $\mathbf{y}$, depending whether b is even or odd, respectively.

Algorithm III Scalar Multiplication.

Input: $k, X \in \mathbb{Z}$ (k to be cleartext, X to be encrypted)
Output: $Z = k \cdot X$.

1: $\mathbf{k} \leftarrow$ standard binary repres. of $|k|$
2: **while** $\mathtt{01}^n \sqsubseteq \mathbf{k}$ for some $n > 1$ **do** ▷ $\sqsubseteq$ . . . substring
3: for longest such n, replace $\mathtt{01}^n$ with $\mathtt{10}^{n-1}\overline{\mathtt{1}}$
4: replace every $\overline{\mathtt{1}}\mathtt{1} \sqsubseteq \mathbf{k}$ with $\mathtt{0}\overline{\mathtt{1}}$
5: $Z \leftarrow 0$
6: **for each** $k_i \in \mathbf{k}$, $k_i \neq 0$
7: $Z \leftarrow Z + k_i \cdot (X \ll i)$ ▷ X shifted, (negated)
8: ▷ use favorite parallel alg.
9: **return** $\mathrm{sgn}(k) \cdot Z$

3.4 Number Comparison & Related Operations

Due to the different nature of the non-standard integer representation for parallel addition, we discuss some other common operations in this representation. Namely, we provide a closer look at number comparison, signum and maximum of two numbers.

3.4.1 Number Comparison

In both of our redundant representations, it shows that direct lexicographic comparison may fail. Let us provide an example for the $(4, \bar{\mathcal{A}}_4)$-representation:

$$(121\bullet)_4 = 25 > 23 = (2\bar{2}\bar{1}\bullet)_4, \tag{22}$$

although $1 < 2$ at the leading position of each number. However, we can subtract the numbers (we can do this in parallel), and then we can compare the result with zero, which works as expected. I.e., the sign of the leading digit determines the sign of the result. We discuss the signum function next.

3.4.2 Signum

Let us recall a method by Bourse et al. [5], which is originally intended to compare two integers $X, Y \in \mathbb{N}_0$: given X and Y in the standard base-β representation as $\mathbf{x}, \mathbf{y} \in \mathcal{A}_\beta^n$, $n \in \mathbb{N}$, where each digit is encrypted with sufficient plaintext space (namely $\pi \geq 3$ and capable of holding $\mathcal{A}_\beta - \mathcal{A}_\beta$), they suggest to subtract $\mathbf{y}$ from $\mathbf{x}$ digit-by-digit, yielding a representation $\mathbf{z} \in (\mathcal{A}_\beta - \mathcal{A}_\beta)^n = \{-\beta + 1, \ldots, \beta - 1\}^n$ of $Z = X - Y \in \mathbb{Z}$. Their method is based on the following observation:

$$\mathrm{sgn}\Big(\sum_{i=0}^{n-1} z_i \beta^i\Big) = \mathrm{sgn}\Big(\sum_{i=0}^{n-1} \mathrm{sgn}(z_i) 2^i\Big), \tag{23}$$

where we obtain a shorter number representation on the right-hand side, hence we can proceed recursively by re-grouping it into larger digits.

Note that the alphabet of $\mathbf{z}$ is symmetric around zero, in addition, for both $\beta = 2, 4$, we have $\bar{\mathcal{A}}_\beta \subsetneq \mathcal{A}_\beta - \mathcal{A}_\beta$. It follows that the method by Bourse et al. can be applied directly with both of our redundant representations $(2, \bar{\mathcal{A}}_2)$ and $(4, \bar{\mathcal{A}}_4)$; find it in Algorithm IV. Importantly, their algorithm is parallelizable using parallel reduction, which takes $O(\log n)$ time in parallel; cf. Figure 1 in their paper.

Algorithm IV Signum over $(\beta, \mathcal{A}_\beta - \mathcal{A}_\beta)$-representation [5].

Input: $(\beta, \mathcal{A}_\beta - \mathcal{A}_\beta)$-representation $\mathbf{z} \in (\mathcal{A}_\beta - \mathcal{A}_\beta)^n$ of $Z \in \mathbb{Z}$, for some $n \in \mathbb{N}$,
Input: plaintext bit-precision $\pi \geq 3$,
Output: $\mathrm{sgn}(Z)$.

1: $\gamma \leftarrow \pi - 1$
2: **function** SignParallelReduce$_\gamma(\mathbf{a}, k)$
3: **if** $k = 1$ **then**
4: **return** $a_0 \gtreqless \pm 1$
5: **for** $j \in \{0, 1, \ldots, \lceil k/\gamma \rceil\}$ **in parallel do**
6: **for** $i \in \{0, 1, \ldots, \gamma - 1\}$ **in parallel do**
7: $s_{\gamma j + i} \leftarrow 2^i \cdot \left(a_{\gamma j + i} \gtreqless \pm 1\right)$ ▷ scale f_1 by 2^i
8: $b_j \leftarrow \sum_{i=0}^{\gamma-1} s_{\gamma j + i}$ ▷ (parallel reduction)
9: SignParallelReduce$_\gamma(\mathbf{b}, \lceil k/\gamma \rceil)$
10: **return** SignParallelReduce$_\gamma(\mathbf{z}, n)$

Remarks In case we want to use this algorithm for number comparison (i.e., we aim at evaluating the signum of $X - Y$), we can save one layer of bootstrapping by changing the identity function, which is applied in the last step of each parallel addition algorithm, to the signum function scaled by an appropriate power of two; cf. line 7 of Algorithm IV. Then we can skip this line during the first recursion level.

Next, we comment on line 8 that parallel reduction might be considered to evaluate the sum. However, since the evaluation does not involve bootstrapping and π is relatively small, we find the possible effect of this option negligible.

Finally, we comment on the prospective usage of signum over the $(2, \bar{\mathcal{A}}_2)$-representation as the activation function in a neural network, where the result gets multiplied by a known scalar. It shows that multiplication of a sample, which encrypts $s \in \bar{\mathcal{A}}_2 = \{-1, 0, 1\}$, by a known scalar w is fairly easy: it is sufficient to copy-paste the sample to the positions of 1's in the binary representation of w and fill the rest with zeros. Note that other representations cannot apply this trick.

Analysis The evaluation of signum on line 7 requires no extra plaintext space (over what is needed for the representation of a's), since signum is a negacyclic function. Indeed, we need the range $[-2^\gamma + 1, 2^\gamma - 1]$, which fits $\pi = \gamma + 1$. Due to line 8 and thanks to the trick on line 7 (i.e., include 2^i into the bootstrapping function), we have $2^{2\Delta} = \gamma$. For the full parallelization, we need n threads, then the algorithm runs in $\lceil \log_\gamma(n) \rceil + 1$ bootstrapping steps. To get the total number

of bootstraps (per digit, respectively), we introduce the following functions:

$$S(n,\gamma) = n + \left\lceil \frac{n}{\gamma} \right\rceil + \left\lceil \frac{n}{\gamma^2} \right\rceil + \ldots + \left\lceil \frac{n}{\gamma^{\lceil \log_\gamma(n) \rceil}} \right\rceil, \tag{24}$$

$$s(n,\gamma) = {}^1\!/_n \cdot S(n,\gamma). \tag{25}$$

3.4.3 Maximum

Next, we outline an approach, how to pick a maximum of two numbers X and Y represented in $(2, \bar{\mathcal{A}}_2)$ or $(4, \bar{\mathcal{A}}_4)$ and encrypted with the multivalue TFHE with $\pi_2 \geq 4$ or $\pi_4 \geq 5$, respectively. Find our method in Algorithms Va and Vb, respectively. In each algorithm, the call of SignParallelReduce$^+_\gamma$ changes line 4 of Algorithm IV, so that it outputs $+1$ for *all non-negative* a_0's (including 0), i.e., it outputs $s \in \{\pm 1\}$. Recall the notation introduced in Section 2.4.1, which is used on lines 6 and 7 of each algorithm.

Algorithm Va Maximum over $(2, \bar{\mathcal{A}}_2)$-representation with $\pi_2 \geq 4$ bits of plaintext space.

Input: $(2, \bar{\mathcal{A}}_2)$-representations $\mathbf{x}, \mathbf{y} \in \bar{\mathcal{A}}_2{}^n$ of $X, Y \in \mathbb{Z}$, for some $n \in \mathbb{N}$,
Output: $\max\{X, Y\}$.

1: $\mathbf{r} \leftarrow \mathbf{x} - \mathbf{y}$ ▷ use favorite parallel alg.
2: $\gamma \leftarrow \pi_2 - 1$
3: $s \leftarrow \text{SignParallelReduce}^+_\gamma(\mathbf{r}, n+1)$
4: **for** $i \in \{0, 1, \ldots, n-1\}$ **in parallel do**
5: **in parallel do**
6: $t \leftarrow (\circ, -1, 0, 1 \| \, 0, 0, 0)[x_i + 2s]$
7: $u \leftarrow (\circ, -1, 0, 1 \| \, 0, 0, 0)[y_i - 2s]$
8: $m_i \leftarrow t + u$
9: **return m**

Algorithm Vb Maximum over $(4, \bar{\mathcal{A}}_4)$-representation with $\pi_4 \geq 5$ bits of plaintext space.

Input: $(4, \bar{\mathcal{A}}_4)$-representations $\mathbf{x}, \mathbf{y} \in \bar{\mathcal{A}}_4{}^n$ of $X, Y \in \mathbb{Z}$, for some $n \in \mathbb{N}$,
Output: $\max\{X, Y\}$.

1: $\mathbf{r} \leftarrow \mathbf{x} - \mathbf{y}$ ▷ use favorite parallel alg.
2: $\gamma \leftarrow \pi_4 - 1$
3: $s \leftarrow \text{SignParallelReduce}^+_\gamma(\mathbf{r}, n+1)$
4: **for** $i \in \{0, 1, \ldots, n-1\}$ **in parallel do**
5: **in parallel do**
6: $t \leftarrow (\circ, -2, -1, 0, 1, 2 \| \, 0, 0, 0, 0, 0)[x_i + 3s]$
7: $u \leftarrow (\circ, -2, -1, 0, 1, 2 \| \, 0, 0, 0, 0, 0)[y_i - 3s]$
8: $m_i \leftarrow t + u$
9: **return m**

Analysis Since the test vectors on lines 6 and 7 of Algorithm Va cannot be overlapped, we need $\pi_2 \geq 4$. Similarly, we need $\pi_4 \geq 5$ in Algorithm Vb. Next, in addition to the requirements of the

selected parallel addition algorithm and $\textsc{SignParallelReduce}^{+}_{\gamma}$, we have: $2+1$ bootstraps per digit in total, $2^{2\Delta_2}=5$ and $2^{2\Delta_4}=10$, respectively, $2n$ threads for the full parallelization, which runs in $1+1$ bootstrapping steps. N.b., using the trick described in *Remarks* for Signum, we save: one bootstrap per digit and one bootstrapping level from the circuit depth.

3.5 Neural Network Evaluation

Scalar multiplication, addition, maximum and signum is a set of operations, which is sufficient for the evaluation of a simple neural network over encrypted input data. Moreover, it offers two used activation functions: signum, and ReLU (*rectified linear unit*), which is defined as $\mathrm{ReLU}(X) = \max\{0, X\}$.

4 Experimental Results

In this section, we put forward the results of two experimental setups: first, we compare all proposed algorithms for addition, and second, we select an addition algorithm and compare it with other operations. For both setups, we provide a brief discussion.

4.1 Experiment Setups

We implemented all presented algorithms for addition in the branch `param-sets` in our experimental library[2], which builds upon the Concrete Library [13] by Zama[3] – a state-of-the-art implementation of the TFHE scheme.

In the first set of experiments, we compare all proposed algorithms for addition of TFHE-encrypted multi-digit integers, whereas in the second set of experiments, we choose and fix an algorithm for addition, and we compare it with algorithms for scalar multiplication, signum and maximum, and we also demonstrate evaluation of a neural network. We run our experiments on a 12-threaded Intel Core i7-7800X processor and for all algorithms, we input 31-bit integers encrypted with TFHE. Note that for all algorithms (except for those for addition with carry), there is still much space for improvement if more threads are available.

Note 4. *We derive the* TFHE *parameter sets for our nine algorithms for addition in our technical paper [21] (they are originally intended for this paper, hence they are using the same letters). According to the LWE Estimator by Albrecht et al. [1, 2], our parameters achieve between* 90 *and* 95 *bits of security, i.e., they are comparable with each other.*

[2] Available at `https://github.com/fakub/parmesan/tree/param-sets`.
[3] `https://zama.ai`

	With carry (Alg. I)			Parallel (Alg. IIa)			Parallel (Alg. IIb)		
Algorithm	I-A	I-B	I-C	IIa-D	IIa-E	IIa-F	IIb-G	IIb-H	IIb-I
Base β	2		4	2			4		
Plaintext bits π	2	2	3	3	4	5	4	5	7
Quad. weight $2^{2\Delta}$	2	3	19	12	12	20	36	36	74
#BS per digit	5	2	$1+1$	$4+1$	$3+1$	$1+1$	$4+1$	$3+1$	$1+1$
#threads	2	2	2	$3(n_2+1)$	$2(n_2+1)$	n_2+1	$3(n_4+1)$	$2(n_4+1)$	n_4+1
Circuit depth	$3n_2$	n_2	n_4	$2+1$	$2+1$	$1+1$	$2+1$	$2+1$	$1+1$
[ms] per BS	40	42	64	61	68	79	76	81	> 346
Cost per bit [ms]	200	84	64	310	270	160	190	160	> 350
Est. best time (31-bit) [ms]	3 700	1 300	1 000	180	200	160	230	240	> 690
Real time [ms] (31-bit integers, 12 threads)	3 740	1 290	1 020	1 190	1 280	686	914	826	$> 1\,780$

Table 1: Comparison of algorithms for addition of multi-digit integers encrypted with TFHE. n_β stands for the digit-length of the integers in base β; *BS* stands for bootstrapping; *Cost per bit* expresses the estimated processor time per 1 bit of inputs; *Estimated best time* assumes enough threads, ideal parallelization and neglects other operations than BS. Experiments were executed on a 12-threaded Intel Core i7-7800X processor. For Algorithm IIb-I, only lower estimates are given; cf. Note 5.

	Addition (Alg. IIa-F)	Scalar Mul. by 121 (Alg. III)	Signum (Alg. IV)	Maximum (Alg. Va)	NN Eval. (Sect. 3.5)
$2^{2\Delta}$	20	20	$\pi - 1 = 4$	$\max\{20, \pi-1, 5\} = 20$	—
#BS per digit	$1+1$	—	$s := s(n_2, \pi-1)$	$s+4$	—
#threads	n_2+1	n_2+1	n_2	$2n_2$	—
Circuit depth	$1+1$	—	$\lceil \log_{\pi-1}(n_2) \rceil + 1$	$\lceil \log_{\pi-1}(n_2) \rceil + 4$	—
Cost per bit [ms]	160	—	$79 \cdot s$	$79 \cdot (s+4)$	—
Est. best time (n_2-bit) [ms]	160	—	$79 \cdot (\lceil \log_{\pi-1}(n_2) \rceil + 1)$	$79 \cdot (\lceil \log_{\pi-1}(n_2) \rceil + 4)$	—
Real time [ms] (31-bit int's, 12 threads)	686	1 407	619	2 028	01:16 [hh:mm]

Table 2: Summary of parallel algorithms for addition, scalar multiplication by 121, signum, maximum and neural network evaluation (cf. Appendix C for its specification) over the $(2, \bar{\mathcal{A}}_2)$ representation. Using fixed TFHE parameters with $\pi = 5$ and $2^{2\Delta} = 20$, running on a 12-threaded Intel Core i7-7800X processor (79 ms per BS). See (25) for the definition of $s(n, \gamma)$.

4.2 Results & Discussion

First, for the comparison of algorithms for addition, we provide an overview of their basic parameters together with their experimental performance results in Table 4.1. The results show that for different usage scenarios, different strategies might show to be the most suitable. Indeed, the redundant representation introduces certain computational overhead, which can be mitigated by sufficient number of threads; a brief discussion of two extreme cases follows.

In case there is only a limited space for the parallelization of bootstrapping and/or if the computation costs are the priority, we would recommend Algorithms I-B or I-C, since they have the lowest cost per bit as well as the total time with only two threads. However, our signum and maximum algorithms could not be used with these addition algorithms due to the small π.

On the other hand, in case there are sufficient parallel resources and the total time is the priority, parallel addition is worth starting from only a few bits: assuming ideal parallelization and our TFHE parameters, Algorithm IIa-F becomes faster than Algorithm I-C from as short as 5-bit integers. Indeed, Algorithm I-C runs $64 \cdot n_4 = 192\,\text{ms}$ (a 5-bit word occupies 3 digits in base 4), whereas Algorithm IIa-F runs $79 \cdot 2 = 158\,\text{ms}$. For the addition of 31-bit (signed) integers, parallel addition using Algorithm IIa-F, with 32 threads and ideal parallelization, is assumed to be already more than $6\times$ faster than the sequential Algorithm I-C. For any other setup, one can easily calculate the best trade-off. However, since future enhancements of the TFHE parameter setup can be expected, hence the ratio between the timings of individual setups may change, our experimental results shall not be percepted as definitive.

Note 5. *With the Concrete Library, the parameters for $\pi = 7$ have shown to be insufficient for the correctness of the results and still they show the worst timings, both in terms of cost per bit and estimated best time. Therefore we decided not to consider this scenario any more, since fixing the parameters would only lead to yet worse results.*

Second, for a comparison of selected algorithm for parallel addition with scalar multiplication, signum and maximum, we have chosen Algorithm IIa-F:

1. it has sufficient $\pi_2 \geq 4$ (required for maximum), and
2. it shows the best times as well as the best cost per bit among all of our parallel algorithms.

For scalar multiplication, we use $k = 121 = (0111'1001\bullet)_2$, which rewrites as $(1000'\bar{1}001\bullet)_2$. Finally, we demonstrate the aforementioned operations on an evaluation of a realistic—yet experimental—neural network for arrhythmia detection (16 inputs, 1 hidden layer of 38 perceptrons, 16 outputs; taken from Mansouri et al. [25]) (find its detailed structure in Appendix C). For an overview of the parameters and respective experimental results, see Table 4.1.

Note 6. *We have chosen neural network evaluation due to the currently limited set of operations, and it serves for demonstration purposes only: the coefficients are not optimized for homomorphic calculations, nor do we perform any rounding or packing. Moreover, as shown by Bourse et al. [4], neural networks achieve fairly good results even when errors occur during their evaluation. Therefore a standard torus-based (unlike our digit-based) encryption with TFHE can be employed to achieve much faster evaluation times.*

5 Conclusion

We proposed several strategies, how selected algorithms for parallel addition of multi-digit integers can be turned into the TFHE-encrypted domain. Our strategies balance between two conflicting motivations: either to limit the plaintext space, or to reduce the number of bootstraps. In addition, we revisited the algorithm for signum by Bourse et al., which turns out to work with the special integer representations that are used for parallel addition, and we presented an algorithm for the maximum of two integers that are given in this representation. As a by-product, we outlined an algorithm for addition with carry with base $\beta = 2$, which uses 2 instead of 5 bootstraps and only slightly tuned TFHE parameters. Based on our experiments, we have concluded that the fastest parallel addition approach can be faster than the fastest sequential one starting from as short as 5-bit integers; for 31-bit integers, it is already more than $6\times$ faster. Finally, on top of the parallel addition algorithm, we proposed a method for scalar multiplication and we demonstrated a combination of our methods on an evaluation of a neural network over encrypted input data.

Future Directions & Open Problems

Our aim is to focus on other multi-digit arithmetic operations in the redundant representation, over encrypted data, with particular respect to their parallelization. E.g., we aim at implementing and optimizing algorithms for multiplication and squaring, on top of that, rounding or reduction modulo 2^k does not show to be as straightforward as in the standard binary representation.

As pointed out in the discussion to Table 4.1, the optimality of the (multivalue) TFHE parameters, given the security level, the plaintext precision and the 2Δ parameter, remains an open question – advances might be expected in the future. Also, we did not consider any packing technique that could possibly greatly improve the evaluation time of a more complex circuit consisting of several independent arithmetic operations.

Acknowledgments

The authors thank Dr. Jan Legerský for his valuable comments. The authors also thank the anonymous referees for their helpful suggestions. This work was supported by the 3IA Côte d'Azur program, reference number ANR19-P3IA-0002, and by the Grant Agency of CTU in Prague, grant No. SGS21/160/OHK3/3T/13.

References

[1] Martin R Albrecht, Benjamin R Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W Postlethwaite, Fernando Virdia, and Thomas Wunderer. Estimate all the {LWE, NTRU} schemes! In *International Conference on Security and Cryptography for Networks*, pages 351–367. Springer, 2018.

[2] Martin R Albrecht, Benjamin R Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W Postlethwaite, Fernando Virdia, and Thomas Wunderer. LWE Estimator. `https://bitbucket.org/malb/lwe-estimator`, 2018.

[3] Algirdas Avizienis. Signed-digit numbe representations for fast parallel arithmetic. *IRE Transactions on electronic computers*, (3):389–400, 1961.

[4] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Annual International Cryptology Conference*, pages 483–512. Springer, 2018.

[5] Florian Bourse, Olivier Sanders, and Jacques Traoré. Improved secure integer comparison via homomorphic encryption. In *Cryptographers' Track at the RSA Conference*, pages 391–416. Springer, 2020.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

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

[8] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.

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

[10] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. *IACR Cryptol. ePrint Arch.*, 2021:91, 2021.

[11] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe. Cryptology ePrint Archive, Report 2021/729, 2021. `https://eprint.iacr.org/2021/729`.

[12] Catherine Y Chow and James E Robertson. Logical design of a redundant binary adder. In *1978 IEEE 4th Symposium onomputer Arithmetic (ARITH)*, pages 109–115. IEEE, 1978.

[13] CONCRETE: Concrete Operates oN Ciphertexts Rapidly by Extending TfhE. `https://concrete.zama.ai/`, 2021.

[14] Léo Ducas and Daniele Micciancio. Fhew: bootstrapping homomorphic encryption in less than a second. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 617–640. Springer, 2015.

[15] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012:144, 2012.

[16] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[17] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the aes circuit. In *Annual Cryptology Conference*, pages 850–867. Springer, 2012.

[18] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210. PMLR, 2016.

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

[20] Malika Izabachène, Renaud Sirdey, and Martin Zuber. Practical fully homomorphic encryption for fully masked neural networks. In *International Conference on Cryptology and Network Security*, pages 24–36. Springer, 2019.

[21] Jakub Klemsa. Setting up efficient tfhe parameters for multivalue plaintexts and multiple additions. Cryptology ePrint Archive, Report 2021/634, 2021. `https://eprint.iacr.org/2021/634`.

[22] Jakub Klemsa and Martin Novotný. WTFHE: neural-netWork-ready Torus Fully Homomorphic Encryption. In *2020 9th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, 2020.

[23] Peter Kornerup. Necessary and sufficient conditions for parallel, constant time conversion and addition. In *Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No. 99CB36336)*, pages 152–156. IEEE, 1999.

[24] Qian Lou and Lei Jiang. She: A fast and accurate deep neural network for encrypted data. *arXiv preprint arXiv:1906.00148*, 2019.

[25] Mohamad Mansouri, Beyza Bozdemir, Melek Önen, and Orhan Ermis. Pac: privacy-preserving arrhythmia classification with neural networks. In *International Symposium on Foundations and Practice of Security*, pages 3–19. Springer, 2019.

[26] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 84–93, 2005.

[27] TFHE: Fast Fully Homomorphic Encryption Library over the Torus. `https://github.com/tfhe/tfhe`, 2016.

[28] Junwei Zhou, Junjiong Li, Emmanouil Panaousis, and Kaitai Liang. Deep binarized convolutional neural network inferences over encrypted data. In *2020 7th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2020 6th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, pages 160–167. IEEE, 2020.

A List of Symbols & Notations

Here we provide a complete list of symbols and notations used throughout this paper:

$\mathbb{N}$ **...** positive integers, i.e., $\{1, 2, 3, \ldots\}$,

$a \div b$ **...** integer division, i.e., $a \div b = \lfloor a/b \rfloor$, $b \neq 0$,

$x \gtrless \pm b$ **...** comparison of x with $\pm b$, $b \in \mathbb{N}$, it outputs $\{-1, 0, +1\}$ as per (2),

$x \equiv \pm b$ **...** comparison of x with $\pm b$, $b \in \mathbb{N}$, it outputs $\{-1, 0, +1\}$ as per (3),

$a \veebar b$ ... logical XOR of a and b,

$\mathsf{MUX}(a, b, c)$ ... evaluates as $a \; ? \; b : c$ (i.e., the standard ternary expression),

$(1, 3, \circ, -2 \| -3, 1)$ ... notation for a custom negacyclic function; cf. Section 2.4.1,

π ... bit-length of the multivalue TFHE message/plaintext space,

2Δ ... logarithm of *quadratic weight*, which is the sum of squared weights; cf. Section 2.4.2,

$(x_{n-1} \dots x_1 x_0 \bullet)_\beta$ ... base-β representation of $\sum_{i=0}^{n-1} \beta^i x_i$,

$\bar{x}$ ... negated integer used in redundant number representation, e.g., $(1\bar{1}\bullet)_2 = 2 - 1 = 1$,

$\mathcal{A}_\beta$ ... alphabet for the standard base-β representation, $\mathcal{A}_\beta = \{0, 1, \dots, \beta - 1\}$,

$\bar{\mathcal{A}}_\beta$ ... alphabet for a redundant representation, given explicitly in Section 2.1.2 for $\beta = 2$, 4.

B The Domain for $x \gtrless \pm b$

We show that the negacyclic function f, $f\colon [-a-b, a+b) \to \{-1, 0, 1\}$, $a \geq b > 0$, defined on the non-negative half of the domain as

$$f(x) = \begin{cases} 0 & x \in [0, b-1], \\ 1 & x \in [b, a], \\ 0 & x \in [a+1, a+b-1], \end{cases}$$

contains the function $x \gtrless \pm b$ on the domain $[-a, a]$.

For $x \in [0, a]$, the claim follows from the definition of f. By the negacyclic property of f, we have $f(x) = -f(x + a + b)$, i.e., for $x \in [-a, 1]$, the function equals

$$-f(x+a+b) = \begin{cases} -1 & x \in [-a, -b], \\ 0 & x \in [-b+1, 1], \end{cases}$$

and the claim follows. Find an example for $a = 6$ and $b = 2$ in Figure 4.

C Structure of the Experimental NN

For demonstration purposes, we evaluated a realistic—yet experimental—neural network for arrhythmia detection (taken from Mansouri et al. [25]). At its input, the network expects 16 signed integers up to 11 bits long, denoted as $\mathbf{x} \in \mathbb{Z}^{16}$. Next, a fully connected hidden layer with 38 perceptrons and with the ReLU activation function follows – we get the intermediate values as

$$\mathbf{p} = \mathrm{ReLU}\big(\mathbf{W_I} \cdot \mathbf{x} + \mathbf{b_I}\big), \tag{26}$$

where $\mathbf{W_I} \in \mathbb{Z}^{38 \times 16}$ stores the weights of the connections and $\mathbf{b_I} \in \mathbb{Z}^{38}$ stands for the vector of biases. Finally, the output layer is also fully-connected and it consists of 16 values calculated as $\mathbf{y} = \mathbf{W_O} \cdot \mathbf{p} + \mathbf{b_O}$. Find the structure depicted in Figure 5.

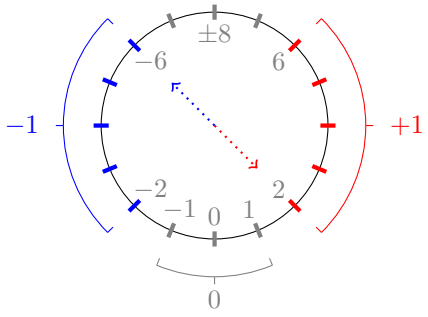

Figure 4: Extension of $x \gtrless \pm 2$ on the domain $[-6, 6]$ to a negacyclic function f. The negacyclic property is outlined with a dotted arrow for $f(-6) = -f(-6 + 8)$.

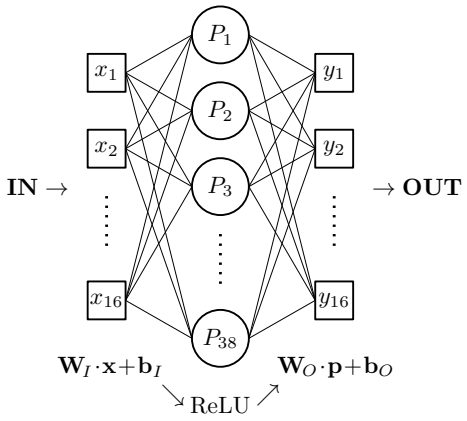

Figure 5: Structure of an experimental neural network for arrhythmia detection: an input layer with 16 inputs, a fully-connected hidden layer with 38 perceptrons using the ReLU activation function, and a fully-connected output layer with 16 outputs (without the activation function).