Updates & Errata

April 2023

New version of Parmesan `v0.1` has been released[1]. This version is rewritten on top of TFHE-rs[2] (`tfhe v0.2`), replacing the dependency on Concrete-core (`concrete-core-experimental v1.0`).

Results of a preliminary comparison of Parmesan `v0.1` and TFHE-rs `v0.2` are given in table below (intended to update Table 5):

		Parmesan `v0.1`		**TFHE-rs** `v0.2`		**Speed-Up**	
Operation	$n =$ #bits	12-thr. [ms]	128-thr. [ms]	12-thr. [ms]	128-thr. [ms]	12-thr.	128-thr.
PBS	–	24	25	–	–	–	–
Add/Sub	32	217	124	137	198	0.63	1.6
Scalar Mul #bits = 32, val's of $k \rightarrow$	4 095	245	167	1 709	2 240	7.00	13.4
	4 096	0.3	0.5	368	572	$\approx \infty$	$\approx \infty$
	4 097	219	113	367	572	1.68	5.1
	805	705	375	1 173	1 618	1.66	4.3
	3 195	710	354	1 294	1 702	1.82	4.8
Mul*	32	8 477	3 046	2 708	2 160	0.32	0.7
Squ*	32	6 003	2 062	2 826	2 532	0.47	1.2
Max (amort.)	32	411	248	377	451	0.92	1.8

For further interpretation, we refer to Section 5.4 (*similar to Concrete-core, for multiplication and squaring of n-bit inputs, TFHE-rs trims the product to n bits, whereas Parmesan outputs the full $2n$-bit output; by little, this affects addition/substraction and scalar multiplication, too).

[1] `https://crates.io/crates/parmesan/versions`
[2] `https://crates.io/crates/tfhe`

PARMESAN: Parallel ARithMEticS over ENcrypted data

Jakub Klemsa$^{(\boxtimes)1,2}$ and Melek Önen[2]

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

`jakub.klemsa@eurecom.fr`, `melek.onen@eurecom.fr`

Abstract

Fully Homomorphic Encryption enables the evaluation of an arbitrary computable function over encrypted data. Among all such functions, particular interest goes for integer arithmetics. In this paper, we present a bundle of methods for fast arithmetic operations over encrypted data: addition/subtraction, multiplication, and some of their special cases. On top of that, we propose techniques for signum, maximum, and rounding. All methods are specifically tailored for computations with data encrypted with the TFHE scheme (Chillotti *et al.*, Asiacrypt '16) and we mainly focus on parallelization of non-linear homomorphic operations, which are the most expensive ones. This way, evaluation times can be reduced significantly, provided that sufficient parallel resources are available. We implement all presented methods in the *Parmesan Library* and we provide an experimental evaluation. Compared to integer arithmetics of the *Concrete Library*, we achieve considerable speedups for all comparable operations. Major speedups are achieved for the multiplication of an encrypted integer by a cleartext one, where we employ special addition-subtraction chains, which save a vast amount of homomorphic operations.

Index terms—Fully homomorphic encryption, Parallelization, Fast arithmetic, TFHE scheme, Benchmarking

1 Introduction

The idea of *Fully Homomorphic Encryption* (FHE), which allows for arbitrary computations over encrypted data, was first proposed by Rivest et al. [40] back in 1978. However, the question of whether such a scheme exists remained open for more than 30 years until 2009 when Gentry [24] gave a positive answer. Although resolved from the mathematical point of view, initial FHE schemes suffered from fairly low efficiency. Since then, the performance of FHE is being constantly improved, either through theoretical advances [25, 8, 12, 26, 22, 13] or with emerging attempts to develop a dedicated hardware [23, 44].

FHE schemes typically allow the evaluation of *addition* and a *non-linear operation* over encrypted data. For addition, this means that there exists operation $\oplus$ over ciphertexts, while for any pair of plaintexts x, y, it holds

$$\mathsf{FHE}.\mathsf{Encr}(x) \oplus \mathsf{FHE}.\mathsf{Encr}(y) \approx \mathsf{FHE}.\mathsf{Encr}(x+y), \tag{1}$$

where $\approx$ means "with high probability, encrypts the same". I.e., $\mathsf{FHE}.\mathsf{Encr}$ is a plaintext $\rightarrow$ ciphertext space additive group homomorphism, up to a randomization of $\mathsf{FHE}.\mathsf{Encr}$ and up to a certain (small) probability of error. The other, non-linear operation can be, e.g., multiplication or *Look-Up Table* (LUT) evaluation.

In principle, FHE enables evaluation of any computable function over encrypted data, e.g., by its decomposition to boolean gates, which may not be very efficient though. For a smooth practical deployment of FHE, we believe that it is important to develop *optimized* homomorphic variants of most common operations, with *basic integer arithmetic* at the first place. Indeed, arithmetic is a fundamental part of most CPUs' instruction sets and integers are one of the primary data types. Since current FHE schemes have a fairly limited plaintext space size, which can only be increased at an unfavorable cost, we build operations upon smaller blocks of data.

In this paper, we put forward tailored and optimized methods for the homomorphic evaluation of basic arithmetic. In addition, we propose homomorphic variants of some other common operations. Our methods are built on top of a particular digit-based integer representation, encrypted with the TFHE Scheme by Chillotti et al. [13]. The TFHE scheme enables a limited number of (very fast) linear operations – these need to be interlaced with another operation referred to as *bootstrapping*, which:

- takes much more time to evaluate (currently tens of milliseconds; depends on parameters),
- is inherently capable of evaluating a custom LUT homomorphically, and
- enables evaluation of circuits of arbitrary depth.

Compared to other FHE schemes, bootstrapping of TFHE is among the fastest, which is the main reason why we choose TFHE. Nevertheless, in our algorithms, the underlying FHE scheme can be easily replaced with another LUT-based FHE scheme, if that scheme shows to be more efficient.

Related Work

Many current use-cases of FHE[1] focus on a single-purpose application, where FHE operations are specifically optimized for this purpose. In particular, there is a lot of interest in cloud-assisted neural network (NN) inference [27, 6, 14], which typically requires expert-level knowledge of both FHE and NN's.

On the other hand, there exists a line of research on *homomorphic compilers*, summarized in [43], which aims at simplifying the homomorphization effort for ordinary developers. Contributions range from a general-purpose transpiler [28] (translates arithmetic operations into many boolean gates, "making them quite slow"), through an approximate-arithmetic-based compiler EVA [20, 17] (whereas we aim at precise arithmetic), to higher-level code optimizations [42].

A scheme known as CKKS [12] (employed in the EVA compiler) enables approximate arithmetic, which is particularly useful in machine learning tasks. However, due to its approximate nature, only a limited precision can be considered correct, therefore, it does not compare directly to our

[1]An updated list of FHE applications can be found at https://fhe.org/fhe-use-cases.

approach. Although there exists a bootstrapped variant of CKKS [11, 10], we are not aware of any implementation of multi-precision arithmetic based on CKKS.

An approach that covers precise integer arithmetics with arbitrary bit-lengths is proposed in [15], further developed in [3], and implemented as part of the *Concrete Library* [19]. Based on a multitude of previous works on homomorphic integers, authors of [3] provide a thorough comparison of state-of-the-art techniques, though mainly focusing on low-level optimizations of bootstrapping and also on finding the best TFHE parameters for selected approaches. For homomorphic arithmetics, they suggest extending the message space by a couple of bits to accommodate the (additive) carry, which allows to evaluate a limited number of additions without the need for bootstrapping. As soon as the carry bits need to propagate, they employ a standard (schoolbook) sequential approach. Authors also propose a parallelizable approach based on the *Chinese Remainder Theorem* (CRT); however, due to its specificities, we do not compare with it.

In our recent study [34], we compare selected sequential and parallel addition algorithms over TFHE-encrypted data: among three sequential and six parallel approaches, we identify the fastest parallel approach, which outperforms the fastest sequential approach starting from 5-bit addends. Since the parallel approach requires a non-standard integer representation, we also demonstrate that other operations like signum and maximum are possible.

Besides integer arithmetics, another important operation is indexing an (encrypted) array with an encrypted index, i.e., evaluation of a big LUT. Two approaches are proposed by Guimarães et al. [30], also studied in [3].

Our Contributions

We propose, implement and evaluate a digit-based integer arithmetic over TFHE-encrypted data, with a particular focus on parallelization, so that the evaluation time is reduced as much as possible. Our methods are based on an algorithm for parallel addition, which we select based on a thorough comparison given in [34]. The list of arithmetic operations includes:

- *Addition/Subtraction:* a basic operation, upon which other operations are built (the underlying algorithm is determined based on the results of [34]). We further identify bootstrap operations that can be saved.
- *Scalar multiplication:* a special case of multiplication, where one integer is unencrypted (demonstrated in [34]). We define a new, presumably hard, computational problem, which is tied with optimization of the number of additions that are called within scalar multiplication (a special type of addition-subtraction chain). Inspired by an approach used in Elliptic Curve Cryptography, we propose a heuristic solution, within which we evaluate small instances of the computational problem, achieving an average improvement of about 20% compared to [34].
- *Multiplication:* the most demanding operation, for which we suggest employing the Karatsuba algorithm to optimize the number of digit-by-digit multiplications, and where we also call the parallel addition algorithm. We discuss and evaluate several aspects of this approach so that the best performance is achieved.
- *Squaring:* a special case of multiplication, where the input is duplicated. We show that a dedicated algorithm for squaring achieves about 30% improvements over multiplication. In addition, we propose a very efficient squaring method for (up to) 3-bit inputs.

We also investigate and optimize other useful operations:

- *Signum:* a fundamental operation for number comparison and other operations (demonstrated in [34]). Compared to [34], we reduce the circuit depth by one, which reduces the number of bootstraps and threads significantly.
- *Maximum:* gives the greater of two encrypted integers (demonstrated in [34]). Not only maximum is improved by the faster signum, but we also propose a new way of evaluation, which only needs half of the threads.
- *Rounding:* rounds an encrypted integer at a given bit-position. The rounding algorithm is non-trivial in the integer representation used by parallel addition.

We accompany each operation with a brief analysis, where we list its requirements (message space size, the ideal number of threads, etc.).

In the experimental part, we present our implementation (in a form of a library) and we compare it with the Concrete Library [19]. Our benchmarks show that for 32-bit encrypted integers, our library achieves speed-ups over Concrete ranging from 1.9× for multiplication on an ordinary 12-threaded server processor, through 7.0× for squaring on a 128-threaded supercomputer's node, to tens of times (and more) for scalar multiplication with selected inputs.

Paper Outline

In Section 2, we recall the TFHE scheme and its supported homomorphic operations: addition and LUT evaluation. We also recall a particular algorithm for parallel addition and its specifics. Next, in Section 3, we revisit and/or suggest new algorithms for basic arithmetic operations, which are suitable for homomorphic evaluation with TFHE, with a particular focus on their parallelization. In Section 4, we revisit and/or suggest other algorithms for comparison-based integer operations: signum, maximum and, rounding. We introduce our implementation and we provide and discuss the results of our benchmarks in Section 5. We conclude the paper in Section 6.

2 Preliminaries

For reference, we first provide a summary of symbols & notation that we use throughout this paper. Then, we recall the TFHE scheme and its homomorphic operations, in particular, we focus on LUT evaluation. Finally, we discuss integer representations and we recall a selected algorithm for parallel addition.

Symbols & Notation

$\mathbb{N}, \mathbb{N}_0$ **. . .** positive and non-negative integers, i.e., $\{1, 2, 3, \ldots\}$ and $\{0, 1, 2, \ldots\}$,

$\mathbb{Z}$ **. . .** the ring of integers,

$\mathbb{R}$, $\mathbb{R}_0^+$ **. . .** real numbers and non-negative real numbers,

$\mathbb{T}$ **. . .** the real torus: $\mathbb{R}/\mathbb{Z}$, i.e., reals modulo 1,

$[a, b)$ **. . .** interval of reals or integers, which contains a and does not contain b,

$x \gtreqless \pm b$ **. . .** comparison of x with $\pm b$, $b \in \mathbb{N}$, it outputs $\{-1, 0, +1\}$ as per (6),

$x \equiv \pm b$ **...** comparison of x with $\pm b$, $b \in \mathbb{N}$, it outputs $\{-1, 0, +1\}$ as per (7),

LUT **...** Look-Up Table,

$(a, b, \circ, c \,\|\, d)$ **...** notation for a custom negacyclic LUT (cf. Section 2.2.2),

π **...** bit-length of the TFHE message space,

$2^{2\Delta}$ **...** the sum of squared weights (aka. *quadratic weight*; cf. Section 2.2.1),

$\mathbf{x} = (x_{n-1} \dots x_1 x_0 \bullet)_\beta$ **...** base-β representation of $X = \sum_{i=0}^{n-1} \beta^i x_i$, where $x_i \in \mathbb{Z}$ and $\beta \in \mathbb{N}$, $\beta > 1$ (cf. Section 2.3.1),

$X = \mathsf{eval}_\beta(\mathbf{x})$ **...** evaluation of representation $\mathbf{x}$ in base β as $X = \sum_{i=0}^{n-1} \beta^i x_i$,

$\bar{x}$ **...** negative digit, $\bar{x} = -x$, $x \in \mathbb{N}$; used in redundant number representations, e.g., $(1\bar{1}\bullet)_2 \sim 2 - 1 = 1$,

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

$\bar{\mathcal{A}}_2$ **...** signed binary alphabet, $\bar{\mathcal{A}}_2 = \{\bar{1}, 0, 1\}$,

MSB/LSB ... Most/Least Significant Bit,

$\mathsf{eval}(\mathsf{AC}_k, X)$ **...** evaluation of addition chain AC_k for integer k and additive group element X into $k \cdot X$ (cf. Section 3.2.1),

ASC^* **...** free-doubling addition-subtraction chain (cf. Section 3.2.2).

2.1 The TFHE Scheme

The TFHE scheme, proposed by Chillotti et al. [13] and recently revisited by Joye [31], is based on a particular variant of the famous *Learning With Errors* (LWE) scheme, first introduced by Regev [39]. 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 plaintext $\mu \in \mathbb{T}$ into ciphertext $\mathbf{c} = (\mathbf{a}, b) \in \mathbb{T}^n \times \mathbb{T}$ (also referred to as the TLWE *sample*) with secret key $\mathbf{s} \in \{0, 1\}^n$ as follows: (i) draw uniformly random mask $\mathbf{a} \overset{\$}{\leftarrow} \mathbb{T}^n$, (ii) draw error term $e \overset{\mathcal{N}}{\leftarrow} \mathbb{T}$ (also referred to as noise) with zero-centered normal distribution $\mathcal{N}$ with standard deviation $\alpha \in \mathbb{R}_0^+$, and (iii) set b as

$$b = \mathbf{s} \cdot \mathbf{a} + \mu + e. \tag{2}$$

In turn, given TLWE sample $(\mathbf{a}, b) \in \mathbb{T}^n \times \mathbb{T}$, decryption evaluates

$$\varphi_{\mathbf{s}}(\mathbf{a}, b) = b - \mathbf{s} \cdot \mathbf{a} = \mu + e, \tag{3}$$

referred to as the *phase function.* Note that it returns the original plaintext *including* the error – we discuss error-free decryption later.

One may observe that *additive homomorphism*, i.e.,

$$\mathsf{Encr}(\mu_1) \oplus \mathsf{Encr}(\mu_2) \approx \mathsf{Encr}(\mu_1 + \mu_2), \tag{4}$$

can be achieved by simple vector addition of TLWE samples. The most important feature of TFHE is so-called *bootstrapping.* The original purpose of bootstrapping is to refresh the magnitude of noise, which must be present in a TLWE ciphertext for security reasons, and which grows with each homomorphic addition. As a convenient byproduct, bootstrapping is inherently capable of homomorphic 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 either be used for the evaluation of logical gates [13], or for the evaluation of multi-value *Look-Up Tables* (LUTs) and/or their compositions [6, 7, 9, 30]. We refer to these variants as the *Binary* TFHE and the *Multi-Value* TFHE, respectively, and we only employ the multi-value variant in this paper[2]. For other practical details on TFHE, we refer to [31].

2.1.1 Multi-Value TFHE

Let $\mathbb{Z}_{2^\pi}$ be the desired message space – each message $m \in \mathbb{Z}_{2^\pi}$ can be represented with π bits. Then, multi-value TFHE encodes message $m \in \mathbb{Z}_{2^\pi}$ into the TLWE plaintext space as $\mu = m/2^\pi \in \mathbb{T}$. The other way around, decoding handles the error from (3) by rounding, i.e., $m' = \lfloor (m/2^\pi + e) \cdot 2^\pi \rceil \in \mathbb{Z}_{2^\pi}$. Note that if $|e| < 1/2^{\pi+1}$, then $m' = m$.

2.2 Homomorphic Operations in Multi-Value TFHE

Combining the encoding of multi-value TFHE and the two homomorphic operations of plain TFHE (i.e., addition and negacyclic LUT evaluation), we obtain a set of homomorphic operations for the $\mathbb{Z}_{2^\pi}$ message space, denoted by $\mathcal{M}$:

- *Addition/Subtraction:* $\mathcal{M} + \mathcal{M} \to \mathcal{M}$ via vector addition/subtraction of TLWE samples;
- *Scalar multiplication:* $\mathbb{Z} \cdot \mathcal{M} \to \mathcal{M}$ via scalar-vector multiplication of a TLWE sample by an integer (equivalent to repeated additions/subtractions; sometimes we refer to both operations simply as *addition*); and
- *Negacyclic* LUT *evaluation:* $\mathsf{LUT}(\mathcal{M}) \to \mathcal{M}$ via TFHE bootstrapping.

2.2.1 Noise Growth during Addition

As outlined in Section 2.1, in the TLWE scheme, a certain amount of noise (error) must be added to the message, and the error term is additive with respect to homomorphic addition. Let us assume a set of fresh(ly bootstrapped) independent samples $\{\mathbf{c}_i\}$, with equal error variance V_0. Then, since error variance is additive with squares of weights, we quantify the error growth after additions using the sum of squared weights:

$$\mathsf{Var}\Big(\sum w_i \cdot \mathbf{c}_i\Big) = \underbrace{\sum w_i^2}_{2^{2\Delta}} \cdot \underbrace{\mathsf{Var}(\mathbf{c}_i)}_{V_0}, \tag{5}$$

where w_i's are integer weights. We refer to $\sum w_i^2$ as the *quadratic weights* and we denote it by $2^{2\Delta}$. E.g., for independent samples x, y, z with equal error variance, we have the quadratic weights of the

[2]In terms of "multi-value bootstrapping" of [9], we consider their first method, for which authors claim: "the output noise is independent of the test polynomial and is the lowest possible" and "only one function can be computed per bootstrapping procedure".

sum $1 \cdot x - 3 \cdot y + 2 \cdot z$ equal to $2^{2\Delta} = 1^2 + 3^2 + 2^2 = 14$. Note that Δ itself is intended to express the additional bit-length of the noise's standard deviation.

2.2.2 (Negacyclic) LUT Evaluation

First, we define a class of functions to make further notation concise and we propose an encoding of these functions into negacyclic LUTs. Then, we outline how additions can be used to evaluate some other, non-negacyclic LUTs. Finally, we introduce a notation for negacyclic LUTs, without explicitly stating them in full.

Threshold Functions & Their Encoding into LUTs Let $b \in \mathbb{N}$. We introduce the following functions:

$$f_b(x) = \begin{cases} -1 & \dots\ x \leq -b, \\ 0 & \dots\ -b < x < +b, \\ +1 & \dots\ +b \leq x, \end{cases} \tag{6}$$

$$g_b(x) = \begin{cases} -1 & \dots\ x = -b, \\ 0 & \dots\ x \neq \pm b, \\ +1 & \dots\ x = +b. \end{cases} \tag{7}$$

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

Recall that LUTs in TFHE are inherently negacyclic, therefore, we need to deal with this limitation. As a usual workaround, an additional bit of padding is added. However, this effectively bloats the message space twice, which in turn induces less efficient TFHE parameters, hence we prefer to avoid that. Instead—as outlined in [33]—we exploit any possible overlap as much as possible, which may lead to message space savings, hence better bootstrapping times. Note that this kind of "overlap optimization" is specific to TFHE – it is also reflected in many of our algorithms, which are—in certain sense—tailored for TFHE.

To encode $x \gtreqless \pm b$ or $x \equiv \pm b$ on a desired domain $[-a, +a]$ (with $a \geq b > 0$) into a negacyclic LUT, the range $[-a - b, a + b)$ shows to be the minimal range for $x \gtreqless \pm b$ and a sufficient range for $x \equiv \pm b$ [34]. For $x \gtreqless \pm b$, we define negacyclic function f, $f\colon [-a - b, a + b) \to \{-1, 0, 1\}$, on the non-negative part of the domain as

$$f(x) = \begin{cases} 0 & x \in [0, b-1], \\ 1 & x \in [b, a], \\ 0 & x \in [a+1, a+b-1]. \end{cases} \tag{8}$$

Such function f contains the function $x \gtreqless \pm b$ on the domain $[-a, +a]$ and the non-negative part of f also serves as a prescription for respective LUT. An analogous approach applies to $x \equiv \pm b$. For more details, we refer to [34].

Evaluating Some Other LUTs Thanks to the cheap additive homomorphism, one may also shift the function by a constant, if this helps to find a negacyclic extension in a smaller domain; an example follows.

Example 1. *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 evaluate the negacyclic* $f(x) - 1$ *over* $\mathbb{Z}_4$ *and add* $+1$ *to the result.*

Moreover, in the plaintext domain of TFHE, i.e., in the torus $\mathbb{T}$, we are not limited to encoding integers – we may also encode fractions. This allows us to evaluate some other non-negacyclic functions; an example follows.

Example 2. $f\colon \mathbb{Z}_4 \to \mathbb{Z}_4$, $f(0) = 0$, $f(1) = 0$, $f(2) = 1$, $f(3) = 1$ *can be evaluated using a shift by* $-1/2$, *as outlined in Example 1.*

In case we consider the first half of $\mathbb{Z}_{2^\pi}$ as positive and the rest as negative (i.e., the standard signed integer representation in computer arithmetics), we may perceive the function from Example 2 as a *non-negativity function* over $\mathbb{Z}_4$. I.e., a function that outputs 1 or 0 if the input is non-negative or negative, respectively.

Notation for Negacyclic LUTs Let $f\colon (\mathbb{Z}_{2^\pi}) \to \mathbb{Z}_{2^\pi}$ be a negacyclic LUT and let $\mathbf{f} \in \mathbb{Z}_{2^\pi}^{2^{\pi-1}}$ be the list of its values in $[0, 2^{\pi-1})$; the rest of f is given by its negacyclicity. For $x \in \mathbb{Z}_{2^\pi}$, referred to as the *selector*, we denote $f(x)$ by $\mathbf{f}[x]$, meaning that x may exceed the index set of $\mathbf{f}$, i.e., the negacyclic extension *is considered.* In case there are some unused function values (i.e., outside of the domain of f), we use the symbol $\circ$, which can be set to, e.g., zeros in $\mathbf{f}$. Finally, in case the value of π is not explicitly given, we use (at most once) the symbol $\|$ to denote the place to be filled with an appropriate number of $\circ$'s. In this case, we *do consider* the negacyclic extension in the suffix; an example follows.

Example 3. *Let* $\pi = 3$, *i.e., we have the message space* $\mathcal{M} = \mathbb{Z}_8$. *The list* $(1, 2, \circ, -3)$ *represents the negacyclic* LUT *given by* $(1, 2, \circ, -3, -1, -2, \circ, 3)$, *whereas the list* $(1, 2 \,\|{-3})$ *represents* $(1, 2, \circ, 3, -1, -2, \circ, -3)$. *With a selector* $-1 = 7$ *in* $\mathbb{Z}_8$, *they evaluate respectively as* $(1, 2, \circ, -3)[-1] = 3$ *and* $(1, 2 \,\|{-3})[-1] = -3$.

2.3 Parallel Arithmetics

The main focus of this paper is the parallelization of arithmetic (and other) operations over encrypted integers. Many of these operations are based on an algorithm for parallel integer addition, which requires a non-standard integer representation. We recall one particular parallel addition algorithm, which we choose based on the results of [34].

2.3.1 Integer Representations

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

$$X = \sum_{i=0}^{n-1} \beta^i x_i =: \mathsf{eval}_\beta(\mathbf{x}). \tag{9}$$

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

For (finite) alphabet $\mathcal{A}$, other than the standard one, with $\mathcal{A} \subset \mathbb{Z}$, we talk about the $(\beta, \mathcal{A})$-*representation.* In particular, parallel addition algorithms typically employ an alphabet that:

1. contains negative digits, represented with bars (i.e., for $d \in \mathbb{N}$, $\bar{d} := -d$),
2. is symmetric around zero (e.g., $\bar{\mathcal{A}} = \{\bar{2}, \bar{1}, 0, 1, 2\}$), and
3. yields a redundant representation.

Point 3 actually states a necessary condition for a parallel addition algorithm to exist, as shown by Kornerup [35].

Example 4. *Let us illustrate redundancy on two different representations of* 2*, using base* $\beta = 4$ *and the aforementioned alphabet* $\bar{\mathcal{A}}$*, as follows:* $(1\bar{2}\bullet)_4 = 1 \cdot 4^1 + (-2) \cdot 4^0 = 0 \cdot 4^1 + 2 \cdot 4^0 = (02\bullet)_4$.

We refer to the $(2, \bar{\mathcal{A}}_2)$-representation, where $\bar{\mathcal{A}}_2 = \{\bar{1}, 0, 1\}$, as the *signed binary* representation. For any alphabet, this kind of representation is also referred to as the *radix-based representation.*

2.3.2 Parallel Addition Algorithm(s)

A family of parameterizable algorithms for parallel addition of multi-digit integers was introduced by Avizienis [2] in 1961. Later in 1978, Chow et al. [16] further improved the meta-algorithm so that it can work with smaller alphabets and even with the minimum integer base $\beta = 2$.

In [34], we compare sequential and parallel algorithms for the addition of TFHE-encrypted integers. We implement three sequential approaches, and two algorithms for parallel addition, namely those using:

- $\beta = 2$, $\bar{\mathcal{A}}_2 = \{\bar{1}, 0, 1\}$ (i.e., signed binary), and
- $\beta = 4$, $\bar{\mathcal{A}}_4 = \{\bar{2}, \bar{1}, 0, 1, 2\}$, respectively.

For both parallel algorithms, we develop three strategies, how each algorithm can be turned into the TFHE-encrypted domain; hence altogether, we compare three + six variants. Based on our experiments, we observe that although parallel approaches introduce a certain computational overhead, the fastest parallel approach outperforms the fastest sequential approach starting from as short as 5-bit integers; for 31-bit integers, it is already more than 6× faster, provided that a sufficient number of threads is available.

For the development of other arithmetic operations to be presented in this paper, we choose the fastest parallel strategy, which uses the signed binary representation; in [34] referred to as *Strategy IIa-F*. We recall this parallel addition method in Algorithm 1. Note that in this paper, we do not further develop nor compare with any sequential approach.

Algorithm 1 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:     $w_i \leftarrow x_i + y_i$
3:     $q_i \leftarrow w_i \gtreqless \pm 2 \vee (w_i \equiv \pm 1 \wedge w_{i-1} \gtreqless \pm 1)$
4:     $z_i \leftarrow w_i - 2q_i + q_{i-1}$                    ▷ (refresh)
5: end for
6: return $\mathbf{z}$
```

Note 1. *In Algorithm 1 on line 3, we abuse notation and we combine the functions $x \gtreqless \pm b$ 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 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 1, we have $0 \vee (+1 \wedge -1) = 0 \vee 0 = 0$.*

2.3.3 Conversions & Other Operations in Signed Binary

The conversion from the standard to the signed binary representation is trivial, since $\mathcal{A}_2 \subset \bar{\mathcal{A}}_2$. Note that this does not hold in general for other signed representations that are used for parallel addition (e.g., $\mathcal{A}_4 \not\subset \bar{\mathcal{A}}_4$), where however parallel addition might be employed. For the opposite direction, a conversion is needed; in addition, from the impossibility result of Kornerup [35], it follows that this conversion *cannot* be parallelized; find more details in Appendix A.

In the signed binary representation, some operations can be implemented fairly straightforwardly: e.g., multiplication, which will be discussed in Section 3.3. Other operations require a more careful approach: e.g., rounding, which will be discussed in Section 4.3. Yet other operations—in particular bit-wise operations—require a conversion to the standard binary, which we leave as future work.

3 Parallel Arithmetics over TFHE-Encrypted Data

In this section, we propose approaches and algorithms for the evaluation of basic arithmetic operations over TFHE-encrypted multi-digit integers, with particular respect to parallelization. In contrast to other works, e.g., [30, 7], we provide our algorithms in the *cleartext* domain, which simplifies their reading and understanding. To turn an algorithm into the encrypted domain, operations are simply replaced with their homomorphic counterparts. Indeed, in our algorithms, either we use basic homomorphic operations of multi-value TFHE (i.e., /weighted/ summation and LUT evaluation), or we rely on algorithms defined previously. Note that this allows us/others to replace the underlying TFHE scheme with another compatible scheme if needed.

Bootstrapping Strategy First, let us commit to a *bootstrapping strategy*: we demand to *always return freshly bootstrapped samples* from all arithmetic operations. I.e., these samples are required to be a direct output of the bootstrapping algorithm, without any further homomorphic additions. Although this is not always needed—in particular in the last step before decryption—we make this guarantee so that the results' correctness is ensured, independent of the operation flow.

Complexity Measure Let us assume:

1. bootstrapping is the *dominant* operation and others are negligible,
2. we have an *unlimited* number of bootstrapping threads,
3. parallelization is *ideal*, i.e., there is no additional orchestration cost.

As the primary complexity measure, we consider the *total running time*, expressed in terms of bootstraps, i.e., the minimal number of consecutive bootstraps in case of ideal parallelization.

Remark 1. *In some extreme cases, we may resort to a different measure, e.g., the total number of bootstraps. Note that the total number of bootstraps is equal to the total running time in the sequential setting (expressed as the number of bootstraps). In practice, it is proportional to the evaluation costs in terms of processor time or electricity consumption.*

3.1 Parallel Addition

First, we focus on the cornerstone arithmetic operation, which is multi-digit integer addition. We recall how the parallel addition algorithm (Algorithm 1) can be turned into the TFHE-encrypted domain. Based on this algorithm, we build other arithmetic operations in the following (sub)sections. We also outline how some non-necessary operations can be avoided in parallel addition.

3.1.1 Parallel Addition in the TFHE-Encrypted Domain

Let us revisit how the selected parallel addition algorithm (Algorithm 1) can be turned into the TFHE-encrypted domain (firstly proposed in [34]). To evaluate w_i and z_i (lines 2 and 4, respectively), additive homomorphism does the job. The value of q_i (line 3) is non-linear in the inputs w_{i-1} and w_i; we illustrate $q_i = q_i(w_{i-1}, w_i)$ in a table in the left-hand side of Figure 1. We suggest to "linearize" the table into a one-dimensional LUT, using $w_{i-1} + 3w_i$ as a selector; we provide an illustration in the right-hand side of Figure 1 (there is indeed a single value in each column). It follows that $q_i(w_{i-1}, w_i)$ can be rewritten as

$$q_i = \left(w_{i-1} + 3w_i \gtreqless \pm 4\right), \tag{10}$$

which allows to construct respective negacyclic LUT, associated to a threshold function (cf. Section 2.2.2). Note that in accordance with our bootstrapping strategy, we apply an additional identity bootstrap on line 4 so that the output consists of freshly bootstrapped samples.

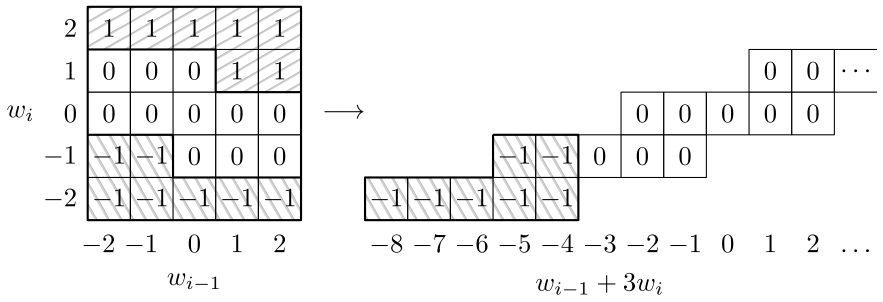


Figure 1: Left-hand side: values of $q_i = q_i(w_{i-1}, w_i)$ as per Algorithm 1. Right-hand side: "linearization" of the table into a one-dimensional LUT, using the selector $w_{i-1} + 3w_i$.

Analysis As shown in [34], we need a message space with $\pi \geq 5$ bits, and we demand quadratic weights $2^{2\Delta} \geq 20$. The algorithm further requires 1 bootstrapping thread per instance (usually per bit of input; a discussion on its optimization follows). It runs in 2 bootstrapping steps, totaling 2 bootstraps per instance.

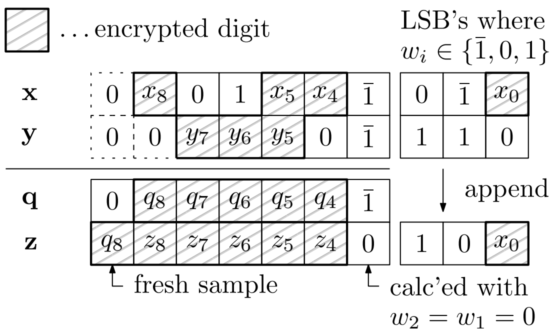
3.1.2 Avoiding Non-Necessary Operations

In case the encrypted digits of two addends $\mathbf{x}$ and $\mathbf{y}$ are not aligned and/or some are unencrypted[3], there may occur non-necessary operations, including bootstraps. Let us discuss these situations with respect to their position, either at the *Least Significant Bit* (LSB) or at the *Most Significant Bit* (MSB).

- *LSB Part:* Suffixes of LSBs of $\mathbf{x}$ and $\mathbf{y}$, where it is guaranteed that all $w_i = x_i + y_i \in \bar{\mathcal{A}}_2$ (e.g., $0 + x$ with x encrypted, or $1 + \bar{1}$, ...), can be separated from both addends before the calculation and then simply appended back. Note that the "missing" separated digits must be considered to be zero for the rest of the calculation.

- *MSB Part:* At the MSB side, the parallel addition algorithm must be performed until the very end due to the left-propagating local carry. On top of that, an additional bit (say of index n) must be prepended: we have $q_n = 0$ and $z_n = x_n + y_n - 2q_n + q_{n-1} = 0 + 0 - 2 \cdot 0 + q_{n-1}$, which is a fresh sample. I.e., there is no need for the refreshal bootstrap of z_n (unlike other z_i's; cf. comment on line 4 of Algorithm 1).

We provide an example in Figure 2. We refer to bits that need to be bootstrapped as *active* bits.



... encrypted digit
LSB's where $w_i \in \{\bar{1}, 0, 1\}$
x
y
q
z
0 x_8 0 1 x_5 x_4 $\bar{1}$ 0 $\bar{1}$ x_0
0 0 y_7 y_6 y_5 0 $\bar{1}$ 1 1 0
0 q_8 q_7 q_6 q_5 q_4 $\bar{1}$
q_8 z_8 z_7 z_6 z_5 z_4 0 1 0 x_0
append
fresh sample
calc'ed with $w_2 = w_1 = 0$

Figure 2: Example of avoiding non-necessary operations (bootstraps) during the addition of integers that are not aligned and/or contain unencrypted digits.

3.2 Scalar Multiplication

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

$$k \odot \mathsf{Encr}(X) \approx \mathsf{Encr}(k \cdot X). \tag{11}$$

By definition, scalar multiplication can be evaluated as $(k-1)\times$ repeated additions of $\mathsf{Encr}(X)$ to itself (later simplified to X). However, we can do better and decrease the number of additions. Let us give a simple example: $4 \cdot X$ can be calculated either as $((X + X) + X) + X$ in three additions, or as $(X + X) + (X + X)$ in just two additions, since $X + X$ can be reused. Hence, our goal is to minimize the number of additions needed to evaluate scalar multiplication – in our case, in the radix-based, TFHE-encrypted domain.

[3]E.g., multiplication of encrypted 3-bit number $x = (x_2x_1x_0\bullet)_2$ by unencrypted $17 = (10001\bullet)_2$ may result in $(x_2x_1x_00x_2x_1x_0\bullet)$, which holds unencrypted zero at the position of 2^3.

Towards Our Method First, we recall an approach adopted in *Elliptic Curve Cryptography* (ECC), namely the so-called *addition(-subtraction) chains*, which aim at minimizing the number of additions (and subtractions) during scalar multiplication over an elliptic curve. Next, we extend the definition of these chains by an assumption that doubling goes for free: unlike ECC, our setup employs a radix-based representation, where the cost of doubling is negligible compared to addition/subtraction. Finally, due to the anticipated intractability of finding the optimal chain of our type, we suggest applying the so-called *window method*. This method splits a particular (signed binary) representation of the actual scalar into a minimum number of sub-scalars of a short, fixed length. For those short scalars, it is feasible to pre-compute the (nearly) optimal chains of our type. Then, we evaluate them and combine the intermediate results with our parallel addition, obtaining the final result of scalar multiplication.

3.2.1 Addition (Subtraction) Chains

As outlined, the number of additions needed for scalar multiplication may differ from approach to approach. Hence, given k, our goal is to find a prescription that evaluates scalar multiplication, while calling the lowest number of additions. In ECC, this problem is formulated in terms of *Addition Chains*, which represent the decomposition of scalar multiplication into additions; let us recall a simplified definition.

Definition 1 (Addition Chain (simplified)). *Let $k \in \mathbb{N}$, $k > 1$. We call the tuple $(1, k_1 \ldots, k_{l-1}, k_l = k)$, $l, k_i \in \mathbb{N}$, an* Addition Chain *for k if $\forall i \in [1, l]$ there $\exists r, s \in [0, i-1]$ such that $k_i = k_r + k_s$. I.e., every element k_i is a sum of some two preceding elements.*

To obtain $k \cdot X$ using an addition chain for k, denoted AC_k, we evaluate AC_k using the same series of additions, but starting from X, instead of 1. We denote the result by $\mathsf{eval}(\mathsf{AC}_k, X) = k \cdot X$.

Many variants of this problem have been proposed and many approaches have been suggested – for a comprehensive overview of these methods, we recommend Chapter 9 of [18]. Here we point out two of them:

- if subtractions are allowed (i.e., $k_i = k_r \pm k_s$ as per Definition 1), we refer to *Addition-Subtraction Chains* (ASC),
- if multiple integers $k^{(0)}, \ldots, k^{(t-1)}$ are to be present in the chain (i.e., there is not only one final k), we refer to *Addition Sequences.*

Downey et al. [21] show that the set of all tuples of the form $(k^{(0)}, \ldots, k^{(t-1)};\ l)$, such that there exists an addition sequence of length l for $\{k^{(0)}, \ldots, k^{(t-1)}\}$, is NP-complete (as a decision problem). It is hence widely believed that also finding the optimal/shortest addition(-subtraction) chain is an intractable task.

3.2.2 Chains with Free Doubling

Between our problem of scalar multiplication and that of ECC, there is a substantial difference: in our case, *doubling goes at a negligible cost*, unlike ECC, where doubling is considered as expensive as addition. Indeed, in our base-2 representation with encrypted digits, doubling melts down to appending an unencrypted zero to the LSB (equivalent to left bit-shift). For this reason, we define another class of ASCs.

Definition 2 (Free-Doubling Addition-Subtraction Chain (ASC^*))**.** *Let $k \in \mathbb{N}$, k is not a power of 2. We call the tuple $([1], [k_1], \ldots, [k_l])$, with $l, k_i \in \mathbb{N}$, k_i odd, a* Free-Doubling Addition-Subtraction Chain *for k if the following holds:*

- $\exists t \in \mathbb{N}$ *such that* $k = 2^t \cdot k_l$,
- $\forall i \in [1, l]$ *there* $\exists r, s \in [0, i-1]$, $t \in \mathbb{N}_0$, *such that* $k_i = \pm k_r \pm 2^t \cdot k_s$.

We consider $[k_i]$ as a class of numbers of the form $2^t \cdot k_i$.

Example 5. *An interesting example of* ASC^* *goes for* $805 =$ `0b1100100101` *– we encourage the reader to try herself before checking the solution*[4].

Hence, the problem of finding the order of additions/subtractions and shifts that lead to $k \odot \mathsf{Encr}(X)$ – with the lowest number of additions/subtractions – melts down to finding the shortest ASC^*, which we assume to be intractable (more research is needed). For this reason, we resort to a heuristic approach.

3.2.3 Rewriting Scalars & Window Method

Due to the anticipated hardness of finding the optimal ASC^* for scalar k, we suggest applying the following approach, inspired by methods of ECC:

1. pre-compute (ideally optimal) ASC^*s for all odd integers of small, fixed bit-length w_l,
2. rewrite the binary representation of k into a signed binary representation, such that there are as long sequences of zeros as possible,
3. apply the sliding window method.

Let us explain each step in detail.

Step 1: Pre-computation of Short ASC^*s We pre-compute ASC^*s for all odd 12-bit integers[5]. The description of our approach is out of the scope of this paper: we leave this for future work and at this moment, we provide the pre-computed ASC^*s "as is". Although we use a brute-force approach, we do not guarantee the optimality, which is rather tricky to show, mainly due to the unlimited power of two within $[k_i]$'s.

Step 2: Rewriting the Scalar To decrease the number of windows in the subsequent window method, it is worth using a signed binary representation for k that not only minimizes the Hamming weight but also maximizes the length of sequences of zeros. For this purpose, we employ the Koyama-Tsuruoka recoding [36]. This recoding minimizes the resulting Hamming weight and on average, it achieves 1.42-bit long sequences of zeros, compared to 1.29-bit for the "traditional" *Non-Adjacent Form* (NAF; [5]).

[4]$([1], [5 = 1 + 1 \cdot 2^2], [25 = 5 + 5 \cdot 2^2], [805 = 5 + 25 \cdot 2^5])$.
Other similar examples are 1 173, 1 209, 1 305, 1 353, 1 377, 1 595, 1 605, 1 695, 1 743, 2 585, 3 129, 3 143, 3 195, 3 205, 3 633, 3 717 and 3 813; some include subtraction, which makes them even more tricky to discover by a pen and paper.

[5]Find our ASC^*s for all odd 12-bit integers in our library [37] in the `assets/asc-12.yaml` file.

$$
\begin{aligned}
&950048719935 \xrightarrow{\textsc{KoyamaTsuruokaRecoding}} \\
&\underbrace{100\bar{1}000\bar{1}0\bar{1}\bar{1}}_{885}\,00\,\underbrace{\bar{1}\bar{1}00\bar{1}0\bar{1}\bar{1}0001}_{-3\,247}\,00000\,\underbrace{1000100000\bar{1}}_{1\,087} \xrightarrow{\textsc{WindowValues\&Shifts}_{12}} \\
&(885, 30), (-3\,247, 16), (1\,087, 0), \text{ for which it holds} \\
&885 \cdot 2^{30} - 3\,247 \cdot 2^{16} + 1\,087 \cdot 2^{0} = 950048719935.
\end{aligned}
$$

Figure 3: Illustration of the window method on top of the Koyama-Tsuruoka recoding.

Step 3: Sliding Window Method Finally, we apply the sliding window method of length 12, which we illustrate in Figure 3 together with the Koyama-Tsuruoka recoding; find a rigorous description of the sliding window method in [18], Chapter 9.1.3.

The Overall Algorithm We provide the overall scalar multiplication method in Algorithm 2. As outlined in the algorithm, any repeated window value can be re-used and also the ASC^*s can be evaluated in parallel. We comment on the final aggregation on line 8 later in Section 3.3.3.

Algorithm 2 Scalar Multiplication.

```
Input: $k, X \in \mathbb{Z}$ ($k$ to be cleartext, $X$ to be encrypted)
Input: $\mathsf{ASC}^*$s of length $l$
Output: $Z = k \cdot X$.
 1: $\mathbf{k} \leftarrow \textsc{KoyamaTsuruokaRecoding}(|k|)$
 2: $(w_i, s_i)_{i=1}^{n_w} \leftarrow \textsc{WindowValues\&Shifts}_l(\mathbf{k})$
                                          ▷ i.e., $|k| = \sum_{i=1}^{n_w} w_i \cdot 2^{s_i}$, $|w_i| < 2^l$
 3: for $i \in \{1, \dots, n_w\}$ in parallel do
 4:     $W_i^{(X)} \leftarrow \mathsf{eval}(\mathsf{ASC}^*_{|w_i|}, X)$        ▷ do not calc. twice for the same $|w_i|$
 5: end for
 6: $Z \leftarrow 0$
 7: for $i = 1 \dots n_w$ do
 8:     $Z \leftarrow Z + \operatorname{sgn}(w_i) \cdot W_i^{(X)} \cdot 2^{s_i}$        ▷ $W_i^{(X)}$ shifted, (negated)
 9: end for
10: return $\operatorname{sgn}(k) \cdot Z$
```

Average Numbers of Additions For 12-bit windows in the Koyama-Tsuruoka recoding, we observe that the average number of additions is 3.10 for ASC^*s, as opposed to 3.88 for the standard double-and-add/sub method, which is suggested in [34] (i.e., about 20% fewer additions). More details are given in Appendix B.

3.3 Multiplication

There exist several (cleartext) algorithms for integer multiplication, most of them extend to algebraic rings, too; for a comprehensive overview, we refer to a thorough survey by Bernstein [4].

Sorted by their asymptotic complexity, below we provide the most famous ones:

- the *schoolbook algorithm*, where every pair of digits gets multiplied, followed by a summation, and which runs in $O(n^2)$,
- the *Karatsuba algorithm* [32], which is based on the *Divide-and-Conquer* strategy and which runs in $O(n^{\log 3})$, and
- the *Schönhage-Strassen algorithm* [41], which is based on a number-theoretic transform and which runs in $O(n \cdot \log n \cdot \log \log n)$.

Although the last one achieves the best asymptotic complexity, it is only worth for huge numbers: e.g., in the GMP Library [29], the threshold `MUL_FFT_THRESHOLD`[6] switches multiplication to the Schönhage-Strassen algorithm for integers longer than high thousands of bits.

Therefore, for the encrypted domain, we do not consider Schönhage-Strassen – instead, we find threshold t_M, starting from which Karatsuba outperforms the schoolbook algorithm.

In the following subsections, we recall the Karatsuba algorithm in the clear, we propose a method for the multiplication of individual encrypted signed bits, and we comment on the final summation in both the schoolbook and Karatsuba algorithm, which also applies to scalar multiplication.

3.3.1 Karatsuba Algorithm

First, let us recall the cleartext version of the Karatsuba algorithm for balanced inputs as Algorithm 3. It follows the *Divide and Conquer* strategy (cf. line 5) and it switches to the schoolbook algorithm if the input length is lower than the threshold t_M (cf. line 2). Indeed, with short inputs, the schoolbook algorithm outperforms Karatsuba; find more details in Appendix C. Note that we do not explicitly recall the schoolbook multiplication algorithm MulSchoolbook$_\beta$, which melts down to pairwise multiplication of individual (signed) digits, followed by a summation.

3.3.2 Multiplication of Individual Encrypted Signed Bits

In our integer representation, digits hold signed bits. Let us outline an algorithm that calculates a product of two encrypted signed bits, using a single LUT evaluation; see Algorithm 4. An illustration of the LUT together with its selector is given in Figure 4. Due to the size of the LUT, we need $\pi \geq 5$.

		$x \cdot y$			$3x + y$		
x	1	−1	0	1	2	3	4
	0	0	0	0	−1	0	1
	−1	1	0	−1	−4	−3	−2
		−1	0	1	−1	0	1
			y				

Figure 4: Values of $x \cdot y$ and those of selector $3x + y$. Find respective LUT in Algorithm 4.

[6]https://gmplib.org/manual/Multiplication-Algorithms, accessed Sep 2022.

Algorithm 3 Karatsuba Multiplication.

Input: $(\beta, \mathcal{A})$-representations $\mathbf{x}, \mathbf{y} \in \mathcal{A}^n$ of $X, Y \in \mathbb{Z}$, for some $n \in \mathbb{N}$,
Input: threshold $t_M \geq 4$,
Output: $X \cdot Y$.

```
1:  function MulKaratsuba_β(r, s)
2:      if $\mathsf{len}(\mathbf{r}) = \mathsf{len}(\mathbf{s}) < t_M$ then
3:          return $\textsc{MulSchoolbook}_\beta(\mathbf{r}, \mathbf{s})$
4:      end if
5:      split $\mathbf{r}, \mathbf{s}$ equally into two parts, s.t. $(\mathbf{r}_1, \mathbf{r}_0) = \mathbf{r}$ and $(\mathbf{s}_1, \mathbf{s}_0) = \mathbf{s}$
                                                          ▷ little-endian representation
6:      $n_0 \leftarrow \mathsf{len}(\mathbf{r}_0) = \mathsf{len}(\mathbf{s}_0)$
7:      in parallel do
8:          $A \leftarrow \textsc{MulKaratsuba}_\beta(\mathbf{r}_1, \mathbf{s}_1)$
9:          $B \leftarrow \textsc{MulKaratsuba}_\beta(\mathbf{r}_0, \mathbf{s}_0)$
10:         $C \leftarrow \textsc{MulKaratsuba}_\beta(\mathbf{r}_1 + \mathbf{r}_0, \mathbf{s}_1 + \mathbf{s}_0)$
11:     end parallel
12:     return $A \cdot \beta^{2n_0} + (C - A - B) \cdot \beta^{n_0} + B$
                                              ▷ calc. using additions & base-$\beta$ shifts
13: end function
14: return $\textsc{MulKaratsuba}_\beta(\mathbf{x}, \mathbf{y})$
```

Algorithm 4 1×1-bit Multiplication in one LUT.

Input: $x, y \in \bar{\mathcal{A}}_2$,
Output: $x \cdot y$.

```
1: return $(0, 0, -1, 0, 1 \| 1, 0, -1, 0)[3x + y]$
```

3.3.3 Summation of Intermediate Results

Both schoolbook and Karatsuba algorithms are followed by a summation of their intermediate results.

For the schoolbook, we illustrate the summation in Figure 5. With each addition, the intermediate value grows one bit to the left (MSB; cf. Figure 2). Therefore, this way, the final result is *exactly* twice as long as the inputs. However, if we decide for some parallelization of the summation, it is worth leaving the last line for the very last step, otherwise, the result gets longer.

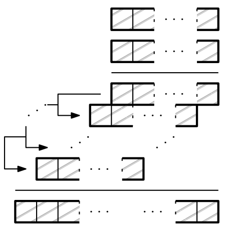


Figure 5: Summation within schoolbook multiplication.

For Karatsuba over n-bit inputs, there are several aspects of the final summation step (cf. line 12 of Algorithm 3) to comment:

1. The result of Karatsuba is *longer than* $2n$ – indeed, the last addition step extends the final result by at least one bit. Note that it also depends on the length of the nested products – these might be already longer than twice their input due to a nested Karatsuba.
2. The value of $-A - B$ can be pre-computed in parallel to the calculation of C, which is more demanding due to the additions $\mathbf{r}_1 + \mathbf{r}_0$ and $\mathbf{s}_1 + \mathbf{s}_0$. Then the value of $C + (-A - B)$ is calculated in the first place.
3. Depending on the length of B, different approaches may apply to minimize the result's length as well as the number of steps:
    - if $|B| = 2n_0$, A and B can be simply concatenated to obtain $A \cdot \beta^{2n_0} + B$, then shifted $C - A - B$ is added;
    - otherwise, B is first added to shifted $C - A - B$, only then shifted A is added.
4. For the first-level Karatsuba (i.e., with all nested schoolbooks), it is worth splitting inputs of odd length such that the LSB part is one bit longer – this approach leads to a shorter addition in the final step. However, for a nested Karatsuba, different approaches may achieve lower bootstrapping complexity; cf. Example 6. Finding the optimal approach for every scenario is out of the scope of this paper.

Example 6. *For a nested Karatsuba, there might be worth another way of splitting odd numbers than the one described in point 4: Let us say we have* $t_M = 16$*. Then, splitting* $31 \rightarrow (16|15)$ *—which is not proposed by point 4 and which calls schoolbook at the LSB part—leads to the concatenation (cf. point 3), and in total, multiplication this way requires* 2 497 *bootstraps. Whereas the proposed way of splitting, i.e.,* $31 \rightarrow (15|16)$*, which calls Karatsuba at the LSB part, requires* 2 531 *bootstraps – mainly due to the additional cost of the* $A \cdot 2^{2n_0} + B$ *addition, instead of their concatenation as in* *the previous case. This gives a counter-example to the odd-number splitting argument, which might not hold in case recursive calls of Karatsuba occur.*

We provide more details on other complexity measures of multiplication (and squaring) later in Section 5.4.1 and (in particular) in Appendix D.

3.4 Squaring

For integer squaring, which is a special case of multiplication, we implement a dedicated algorithm. Similarly to Karatsuba multiplication, we employ the divide and conquer strategy; find our method for squaring in Algorithm 5. For the threshold t_S (line 2), we obtain the value $t_S = 4$ for our setup, using an approach analogical to multiplication (cf. Appendix C). We comment on the schoolbook squaring algorithm (line 3) later.

Algorithm 5 Squaring via Divide-and-Conquer.

```
Input: $(\beta, \mathcal{A})$-representation $\mathbf{x} \in \mathcal{A}^n$ of $X \in \mathbb{Z}$, for some $n \in \mathbb{N}$,
Input: threshold $t_S \geq 4$,
Output: $X^2$.
 1: function SquDivNConq$_\beta$($\mathbf{r}$)
 2:     if $\mathsf{len}(\mathbf{r}) < t_S$ then
 3:         return SquSchoolbook$_\beta$($\mathbf{r}$)
 4:     end if
 5:     split $\mathbf{r}$ equally into two parts, s.t. $(\mathbf{r}_1, \mathbf{r}_0) = \mathbf{r}$
                                                        ▷ little-endian representation
 6:     $n_0 \leftarrow \mathsf{len}(\mathbf{r}_0)$
 7:     in parallel do
 8:         $A \leftarrow$ SquDivNConq$_\beta$($\mathbf{r}_1$)
 9:         $B \leftarrow$ SquDivNConq$_\beta$($\mathbf{r}_0$)
10:         $C \leftarrow$ MulKaratsuba$_\beta$($\mathbf{r}_1, \mathbf{r}_0$)
11:     end parallel
12:     return $A \cdot \beta^{2n_0} + C \cdot \beta^{n_0+1} + B$          ▷ additions & base-$\beta$ shifts
13: end function
14: return SquDivNConq$_\beta$($\mathbf{x}$)
```

Compared to multiplication on a duplicated input, our squaring algorithm evaluates fewer bootstraps in fewer steps, which is mainly achieved thanks to:

- fewer terms to be evaluated:
  - two additions on line 10 of Algorithm 3 are not evaluated on line 10 of Algorithm 5,
  - instead of $C - A - B$ on line 12 of Algorithm 3, there is only C on line 12 of Algorithm 5; and
- more efficient squaring of short inputs in case of signed binary representation.

Squaring of Short, Signed Binary Inputs For short, signed binary input $\mathbf{x}$ (namely 2- or 3-bit with $\pi = 5$), we suggest to calculate individual bits of the resulting square directly and in parallel. The idea is to evaluate homomorphically $X = \mathsf{eval}_2(\mathbf{x})$ as per (9), which melts down to scalar multiplications and additions of TFHE samples (i.e., no bootstrap is needed). To evaluate a bit of $Y = X^2$, we use X as a selector into a dedicated LUT; find an illustration in Table 1, where columns represent these LUTs. For the full algorithm, we refer to Appendix E, Algorithm 10.

Table 1: Bits of $Y = X^2$ and respective selector X. Columns y_i are intended to be encoded into LUTs.

X	bits of $Y = X^2$						X^2
	y_5	y_4	y_3	y_2	y_1	y_0	
0	0	0	0	0	0	0	0
±1	0	0	0	0	0	1	1
±2	0	0	0	1	0	0	4
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
±7	1	1	0	0	0	1	49

Note that in the signed binary, 3 bits may encode $X \in [-7, 7]$, and the output is up to 6-bit. For 2-bit inputs, we only calculate the output bits up to y_3. Also note that the bit at 2^1 position (i.e., y_1) is always zero, which stems from the fact $a^2 \bmod 4 \in \{0, 1\}$. Hence, for 2- and 3-bit inputs, we evaluate 3 and 5 LUTs, respectively, and we obtain the result in a single bootstrapping step. We provide more details on other complexity measures of squaring (and multiplication) later in Section 5.4.1 and (in particular) in Appendix D.

Recall that we have $t_S = 4$ and we use the signed binary, i.e., $\textsc{SquSchoolbook}_\beta$ (on line 3 of Algorithm 5) is fully implemented via this method for squaring of short inputs.

4 Signum-Based Operations over TFHE-Encrypted Data

In this section, we put forward some other, frequently used, signum-based operations in the signed binary representation, while bearing in mind the limited set of available homomorphic operations over the encrypted digits. Namely, we present parallel algorithms for *signum* and *maximum* (improved versions of [34]), and we introduce a new algorithm for *rounding* at a selected digit position.

In principle, these algorithms are based on number comparison. However, in the signed binary representation, the lexicographic comparison may fail[7]. Therefore, we suggest reducing the problem of number comparison to signum: we subtract the numbers (in parallel) and we compare the result with zero. This works in the signed binary as expected, i.e., the sign of the leading bit determines the sign of the result.

4.1 Signum

A method for comparison of two integers, given as a series of encrypted digits, was firstly proposed by Bourse et al. [7]. Later, we adjusted this method to signed integer representations in [34]; we recall it in Algorithm 6.

[7]As an example, $(\mathbf{0}11\bullet)_2 = 3 > 2 = (\mathbf{1}\bar{1}0\bullet)_2$, although $0 < 1$ at the leading position of each number.

Algorithm 6 Signum over $(\beta, \mathcal{A}_\beta - \mathcal{A}_\beta)$-representation ([7]; modified).

```
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: message space with bit-length $\pi \geq 3$, such that $2^{\pi-1} \geq \beta$,
Output: $\mathrm{sgn}(Z)$.
1: $\gamma \leftarrow \pi - 1$
2: function SgnParalReduce$_\gamma(\mathbf{a})$
3:     $k \leftarrow \mathsf{len}(\mathbf{a})$
4:     if $k = 1$ then
5:         return $a_0 \gtreqless \pm 1$
6:     end if
7:     for $j \in \{0, 1, \ldots, \lceil k/\gamma \rceil - 1\}$ in parallel do
8:         for $i \in \{0, 1, \ldots, \gamma - 1\}$ in parallel do
9:             $s_{\gamma j+i} \leftarrow 2^i \cdot \left(a_{\gamma j+i} \gtreqless \pm 1\right)$          ▷ scale $f_1$ by $2^i$
10:        end for
11:        $b_j \leftarrow \sum_{i=0}^{\gamma-1} s_{\gamma j+i}$
12:    end for
13:    return SgnParalReduce$_\gamma(\mathbf{b})$
14: end function
15: return SgnParalReduce$_\gamma(\mathbf{z})$
```

We propose an improvement, which only works in the signed binary representation: we suggest skipping the bootstrapped (and scaled) comparison $a_{\gamma j+i} \gtreqless \pm 1$ on line 9 of Algorithm 6 in the first level of recursion since we have already $a_{\gamma j+i} \in \{\bar{1}, 0, 1\}$. Instead, $a_{\gamma j+i}$'s get directly scalar-multiplied by 2^i's and aggregated into b_j, which—compared to [34]—saves one level of bootstrapping and reduces the number of threads by a factor of about four.

Next, note that the comparison function on line 5 can be replaced with another function if needed – n.b., the value of a_0 is only guaranteed to have the same sign as the top-level input. E.g., one may compute the non-negativity function as outlined in Example 2, which is useful for number comparison.

Analysis The evaluation of $a \gtreqless \pm 1$ on lines 5 and 9 (i.e., signum of a) requires no extra plaintext space (over what is needed for the representation of a's) since signum is already negacyclic. Indeed, we need the range $[-2^\gamma + 1, 2^\gamma - 1]$, which perfectly fits within $\pi = \gamma + 1$ bits.

The aforementioned optimization (line 9 in the first level of recursion) mandates $2^{2\Delta} \geq (2^0)^2 + \ldots + (2^{\gamma-1})^2$, which equals 85 for $\pi = 5$. Note that this is the largest value of $2^{2\Delta}$ within Parmesan.

For the full parallelization, we need $\lceil n/\gamma \rceil$ threads (bootstrapping starts from the second level of recursion; as opposed to [34], which therefore requires n threads), and the algorithm runs in $\lceil \log_\gamma(n) \rceil$ bootstrapping steps. The total number of bootstraps can be expressed as

$$S_\gamma(n) := \left\lceil \frac{n}{\gamma} \right\rceil + \left\lceil \frac{n}{\gamma^2} \right\rceil + \ldots + \left\lceil \frac{n}{\gamma^{\lceil \log_\gamma(n) \rceil}} \right\rceil. \tag{12}$$

4.2 Maximum

We present an improved version of the method [34] for maximum of two integers X and Y, represented in the signed binary representation and encrypted with $\pi \geq 5$; find it in Algorithm 7. The function $\textsc{SgnParalReduce}_\gamma^+$ on line 3 customizes line 5 of Algorithm 6, so that it evaluates the non-negativity function (cf. Example 2). Recall the notation introduced in Section 2.2.2, which is now used on line 5 of the algorithm. We illustrate the LUT creation in Figure 6.

$\max\{x,y\}_i$

x_i \ y_i	$x<y$: $\bar{1}$	$x<y$: 0	$x<y$: 1	$x \geq y$: $\bar{1}$	$x \geq y$: 0	$x \geq y$: 1
1	$\bar{1}$	0	1	1	1	1
0	$\bar{1}$	0	1	0	0	0
$\bar{1}$	$\bar{1}$	0	1	$\bar{1}$	$\bar{1}$	$\bar{1}$

$s{+}2x_i{+}6y_i$

x_i \ y_i	$\bar{1}$	0	1	$\bar{1}$	0	1
1	-4	2	8	-3	3	9
0	-6	0	6	-5	1	7
$\bar{1}$	-8	-2	4	-7	-1	5

Figure 6: Values of i-th bit of $\max\{x,y\}$ for both cases $x<y$ and $x \geq y$, and those of respective selector $s+2x_i+6y_i$, where $s=(x \geq y)$, i.e., the non-negativity function.

Algorithm 7 Maximum over $(2,\bar{\mathcal{A}}_2)$-representation with $\pi \geq 5$ 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: $\gamma \leftarrow \pi - 1$
3: $s \leftarrow \textsc{SgnParalReduce}_\gamma^+(\mathbf{r})$
4: for $i \in \{0,1,\ldots,n-1\}$ in parallel do
5:     $m_i \leftarrow \big(0,0,0,1,1,\bar{1},1,0,1,1 \| (\bar{1},\bar{1}),\bar{1},0,\bar{1},1,0,\bar{1}\big)[s+2x_i+6y_i]$
6: end for
7: return $\mathbf{m}$
```

Implementation Remark Note that for $\pi = 5$, there is a negacyclic overlap of two values within the LUT on line 5: $1,1$ before $\|$ is directly followed by its own negacyclic image $\bar{1},\bar{1}$, which is therefore given in parentheses.

Analysis In addition to the requirements of subtraction and those of $\textsc{SgnParalReduce}_\gamma^+$ (n.b., $\mathbf{r}$ is one bit longer than $\mathbf{x}$ and $\mathbf{y}$), we have: one bootstrap per bit (as opposed to three bootstraps in [34]), $2^{2\Delta} \geq 1^2+2^2+6^2 = 41$ due to the selector on line 5, and we need n threads for the full parallelization (vs. $2n$ threads in [34]). In total, maximum runs in $2+\lceil \log_\gamma(n+1) \rceil + 1$ bootstrapping steps with the total number of $2n + S_\gamma(n+1) + n$ bootstraps; cf. (12) for the definition of $S_\gamma(\cdot)$.

4.3 Rounding

Integer rounding operation at a given position (within its binary representation) can be expressed as function R of two inputs: integer $X \in \mathbb{Z}$ to be rounded, and position $i \in \mathbb{N}$ to hold the last non-zero bit. The function can be written as

$$R(X, i) \coloneqq \left\lfloor \frac{X}{2^i} \right\rceil \cdot 2^i = \dots, \tag{13}$$

for $X = (x_{n-1} \dots x_i x_{i-1} \dots x_0 \bullet)_2$ also as

$$\dots = (x_{n-1} \dots x_i 0 \dots 0 \bullet)_2 + \lfloor \underbrace{(\bullet x_{i-1} \dots x_0)_2}_{r} \rceil \cdot 2^i. \tag{14}$$

With the standard binary alphabet, the value of $\lfloor r \rceil = \lfloor (\bullet x_{i-1} \dots x_0)_2 \rceil$ equals to x_{i-1}. Indeed, if $x_{i-1} = 0$, then the remainder $r = (\bullet 0 x_{i-2} \dots x_0)_2$ is always lower than $1/2$, conversely for $x_{i-1} = 1$, $r = (\bullet 1 x_{i-2} \dots x_0)_2 \geq 1/2$.

However, with the signed binary alphabet, the leading bit x_{i-1} does not determine how r compares to $1/2$. In addition, such r ranges in the interval $(-1, 1)$, unlike $[0, 1)$ for the standard binary alphabet, therefore, we need to compare with both $\pm 1/2$. Altogether nine combinations occur for $x_{i-1} \in \bar{\mathcal{A}}_2$ and for signum of the rest of r, denoted $s \coloneqq \operatorname{sgn}\big((\bullet x_{i-2} \dots x_0)_2\big)$. The resulting value of $\lfloor r \rceil$ as well as that of respective selector $2x_{i-1} + s$ are depicted in Figure 7. Find our method in Algorithm 8.

$\lfloor (\bullet x_{i-1} \dots)_2 \rceil$

x_{i-1} \ s	-1	0	1
1	0	1	1
0	0	0	0
$\bar{1}$	$\bar{1}$	0	0

$2x_{i-1} + s$

x_{i-1} \ s	-1	0	1
1	1	2	3
0	-1	0	1
$\bar{1}$	-3	-2	-1

Figure 7: Values of $\lfloor r \rceil = \lfloor (\bullet x_{i-1} \dots x_0)_2 \rceil$ and those of respective selector $2x_{i-1} + s$, where $s = \operatorname{sgn}\big((\bullet x_{i-2} \dots x_0)_2\big)$.

Analysis Rounding calls signum, evaluates a LUT, and runs parallel addition, while the LUT evaluation requires neither larger π nor $2^{2\Delta}$ than any of those operations. Altogether, for $i > 1$, rounding requires $\max\{\lceil i-1/\gamma \rceil, n - i\}$ threads, it runs in $\lceil \log_\gamma (i-1) \rceil + 1 + 2$ bootstrapping steps, and in total $S_\gamma(i-1) + 1 + 2 \cdot (n-i)$ bootstraps are called; cf. (12).

5 Implementation & Experimental Results

In this section, we introduce our library for parallel arithmetics over TFHE-encrypted data. Then, we comment on its dependency on the Concrete Library and we also compare the abilities of these libraries in their current versions. Next, we outline an experiment design, covering the choice of parameters, inputs, and hardware. Finally, we put forward the results of our benchmarks and we conclude with a brief discussion.

Algorithm 8 Rounding over $(2, \bar{\mathcal{A}}_2)$-representation.

```
Input: $(2, \bar{\mathcal{A}}_2)$-representation $\mathbf{x} \in \bar{\mathcal{A}}_2{}^n$ of $X \in \mathbb{Z}$, for some $n \in \mathbb{N}$,
Input: rounding position $i \in \mathbb{N}$,
Output: $R(X, i)$ as per (13).
1: if $i > n$ then
2:     return 0
3: end if
4: if $i = 1$ then
5:     $t \leftarrow x_0$
6:     go to line 11
7: end if
8: $\gamma \leftarrow \pi - 1$
9: $s \leftarrow$ SgnParalReduce$_\gamma((x_{i-2}, \ldots, x_0))$          ▷ $(x_{i-2}, \ldots)$ might be empty
10: $t \leftarrow (0, 0, 1, 1 \| \bar{1}, 0, 0)[2x_{i-1} + s]$
11: $\mathbf{u} \leftarrow (x_{n-1} \ldots x_i \bullet)_2 + (t \bullet)_2$          ▷ use favorite parallel alg.
12: return $\mathbf{u} \ll i$
```

5.1 The PARMESAN Library

We implement all operations, presented in the previous sections, in the PARMESAN Library [37]. Parmesan is an experimental library based on an existing implementation of TFHE – the Concrete Library [19], which we discuss later. Parmesan, as well as Concrete, are written in Rust[8] and they are compatible with the Rust's ecosystem, i.e., they can be easily added to a custom project via a standard Rust dependency.

To make the starting point smooth, our library goes with a simple demo (in the `README` file), which includes:

- TFHE parameter initialization, which loads a hard-coded parameter set;
- creation of User's and Cloud's scopes, which also generates respective keys;
- digit-by-digit encryption of integers a and b, given in a (signed) base-2 representation;
- homomorphic addition $\mathsf{Encr}(a) \oplus \mathsf{Encr}(b)$;
- decryption of the result; and
- final check if $\mathsf{Decr}\big(\mathsf{Encr}(a) \oplus \mathsf{Encr}(b)\big) = a + b$.

All supported operations can be found in the `ParmArithmetics` trait, which is implemented for both Parmesan ciphertexts *and* for signed 64-bit integers (Rust type `i64`).

5.2 The Concrete Library

Among existing implementations of TFHE, we choose the Concrete Library [19] by Zama[9]: Concrete is open-source[10], is actively developed, implements state-of-the-art techniques, and also long-lasting

[8]https://rust-lang.org
[9]https://zama.ai
[10]Under the BSD 3-Clause Clear License.

support can be expected. At the time of writing, the latest release of Concrete is the `beta-2` version of `v0.2.0`, which we later denote as $\texttt{v0.2}_\beta$. In the beta version, many features are not stabilized yet and further improvements are expected to come with the full version (including a certain level of parallelization).

The Concrete library is written in Rust, where bundles of code are referred to as "crates". The `concrete` crate covers more than the implementation of TFHE (in the `concrete-core` sub-crate) – it consists of other three sub-crates: `concrete-boolean`, `concrete-shortint` and `concrete-int`, which respectively implements booleans, 2- to 7-bit unsigned integers and multi-precision unsigned integers, including various homomorphic operations for each type.

5.2.1 Relation to Parmesan

Internally, Parmesan's TFHE ciphertexts and respective homomorphic operations employ structures and functions of `concrete-core`. On top of these TFHE-level operations, integer arithmetic is built from the scratch: starting from the signed, radix-based representation, until the implementation of various arithmetic operations and their parallelization.

5.2.2 Concrete's Arithmetics

In `concrete-int`, various arithmetic as well as bit-level operations are implemented over radix-based integer representations with selected power-of-two bases. Currently, a limited set of operations is implemented also for the CRT-based[11] representation (addition and multiplication; appears to be under development). In our experiments, we focus solely on the radix-based representation, which is by its nature closer to Parmesan. In addition, CRT-based representation cannot be used to mimic standard computer arithmetics, which operate $\bmod 2^n$, unlike CRT, which operates modulo a product of coprime integers; for more details on CRT-based arithmetics, we refer to [3].

Remark 2. *Given base $\beta = 2^k$ and digit-length l, the radix-based arithmetic of Concrete is equivalent to the standard unsigned kl-bit integer arithmetic. For each of the l digits, encrypted with* TFHE*, the cleartext space actually consists of two parts: a k-bit* message *part, which covers the standard base-β alphabet, and a* carry *part, which effectively extends the standard alphabet by a couple of additional bits. This allows performing a certain number of additions without the need for bootstrapping. In our experiments with Concrete, we run multiple additions until we reach the first bootstrap and in the results, we amortize the cost.*

Multiplication is implemented using the standard schoolbook algorithm without any parallelization and for squaring, there is no special function. For scalar multiplication, there is no optimization in terms of free doubling (we verify this by calling multiplication by $4\,096 = 2^{12}$), nor in terms of addition-subtraction chains.

5.2.3 Parmesan's vs. Concrete's Arithmetics

We provide a comparison of Parmesan's and Concrete's operations in Table 2. Usage-wise, the biggest difference is that Concrete mimics the behavior of native unsigned integer types (i.e., that of the ring $\mathbb{Z}_{2^{kl}}$), whereas Parmesan natively supports negative integers. Regarding the precision bound, this is rather a matter of implementation and both libraries could be easily extended.

[11] Chinese Remainder Theorem.

Table 2: The current state of implementation of arithmetics in Parmesan (experimental library) and in Concrete `v0.2`$_\beta$ (some features not yet fully implemented in beta).

Feature	**Parmesan**	**Concrete** `v0.2`$_\beta$
Radix-based representation	✓ (signed, unlimited)	✓ (unsigned, mod 2^{kl})
CRT-based representation	✗	(✓)
Addition/subtraction, multiplication	✓(parallel)	✓(sequential)
ASC*s for scalar multiplication	✓	✗
Karatsuba multiplication	✓	✗
Dedicated squaring	✓	✗
Bit operators	✗	✓
Signum, maximum, rounding	✓	✗

5.3 Experiment Setup

Let us discuss particular choices of parameters, inputs, and hardware.

5.3.1 Choice of Parameters

In Parmesan, we need 5 bits of message space and we do not need any padding. For this purpose, we choose Concrete's parameters named `PARAM_MESSAGE_2_CARRY_3`: there are 2 bits for the message and 3 extra bits for the carry, altogether 5 bits. Although there is no parameter corresponding to $2^{2\Delta}$ in Concrete, we did not encounter any error during any of our experiments or tests of Parmesan with this parameter choice, therefore, we consider our choice adequate. All parameters in Concrete are claimed to be chosen with the (expected) level of 128-bit security, which we verify with the `lattice-estimator`[12] [1].

In Concrete, we use the default builder for 2-bit unsigned integers, upon which we build longer integers. Regarding the digit's bit-length, there is no clear recommendation, however, our experiments with parallel addition algorithms [34] as well as an example implementation of the Game of Life[13] tend to prefer shorter message space.

5.3.2 Choice of Inputs

We choose to benchmark 4-, 8-, 16-, and 32-bit values with Addition/Subtraction, Multiplication, and Squaring. For Signum, Maximum, and Rounding, we only benchmark 32-bit numbers to spot the effect of recursion.

For Scalar Multiplication, we choose to verify the following scalars: 4 095, 4 096, 4 097, 805, and 3 195. For 4 096, there shall be no operation needed in any of the bases: 2 (Parmesan), 4 (our setup of Concrete), 8, or 16. For 4 095 and 4 097, we aim at observing, whether one operation is used in both cases, i.e., whether both ways $4\,095 = 2^{12} - 1$ and $4\,097 = 2^{12} + 1$ are used. If this is not the case, we would observe a big gap for 4 095, since its Hamming weight is 12, as opposed to 2 for

[12] https://github.com/malb/lattice-estimator
[13] https://www.zama.ai/post/the-game-of-life-rebooted-with-concrete-v0-2, accessed Sep 2022.

4 097. As outlined in Example 5, 805 has a very efficient addition chain of just 3 additions, a similar property goes also for 3 195.

5.3.3 Choice of Hardware

For our experiments, we choose two machines:

- an experimental server with a 12-threaded Intel Core i7-7800X processor (EURECOM's internal machine), and
- a supercomputer's node with two 64-threaded AMD EPYC 7543 processors (operated by e-INFRA CZ[14]).

Note that the 128-threaded machine has a sufficient number of threads to achieve the full parallelization for most of the operations (unlike, in particular, the multiplication of long integers).

5.4 Results & Discussion

With chosen parameters, inputs, and hardware, we benchmark Parmesan using our dedicated experimental tool[15]. Since our main aim is to benchmark Parmesan on a highly multi-threaded processor, we accompany the code with scripts tailored for the *Portable Batch System* (PBS), which is a queuing system of many super-computing infrastructures, including e-INFRA CZ.

5.4.1 Observed Quantities

In our experiments, we primarily focus on the *total running time* (as per our Complexity Measure; cf. Section 3), and we also approximately measure the *processor load*. Besides that, we analytically evaluate other quantities:

- *(bootstrapping) circuit depth*: the minimal number of consecutive bootstraps in case of ideal parallelization,
- *total number of bootstraps (aka. #PBS)*,
- *ideal number of threads*: the minimum number of threads required for ideal parallelization, and
- *efficiency of CPU/thread usage*: the total number of available bootstrapping slots (i.e., the ideal number of threads multiplied by the circuit depth) divided by the total number of bootstraps called.

Example 7. *Let us evaluate these analytical quantities on an example of* 16-bit multiplication *(cf. Algorithm 3). The calculation spreads into three pools of threads to calculate the values of A, B, and C (cf. lines 8–10), using two instances of 8-bit schoolbook multiplication for A and B, and two 8-bit additions followed by a 9-bit schoolbook multiplication for C. For each 8-bit multiplication, we need 64 threads for pairwise multiplications, then we call $7\times$ addition (cf. Figure 5), each with 8 active bits (i.e., 8 threads in 14 steps). For 9-bit multiplication, we need 81 threads (multiplication) followed by 16×9 threads (summation).*

[14]https://e-infra.cz/en

[15]https://github.com/fakub/bench-parmesan

The calculation of A and B is followed by their addition (2×16 threads), then A+B is subtracted from C (2×18 threads) and finally $C-(A+B)$ is added to concatenated $A\|B$ (n.b., their length allows that; 2×24 threads). In total, we have 725 bootstraps in 23 steps and it shows that 81 threads are necessary & sufficient for the full parallelization; see Table 3, where columns A, B and C are later re-used for other calculations. Efficiency evaluates to $^{725}/_{81\cdot 23} \approx 38.9\%$. *Find other bit-lengths, also for squaring, in Appendix D.*

Table 3: A suggestion of thread scheduling for the calculation of intermediate values A, B, and C, followed by their aggregation, in 16-bit Karatsuba multiplication, which splits each input into two 8-bit parts. Using 81 threads in 23 steps, totalling 725 bootstraps.

A	B	C	Total #thr's	*Comment*
64	–	8|8	80	C: $\mathbf{r}_1+\mathbf{r}_0 \mid \mathbf{s}_1+\mathbf{s}_0$
–	64	8|8	80	
–	–	81	**81**	C: 9-bit pairwise mul.
8	8	9	25	A, B: 8-bit schoolbook summation (14 rows);
8	8	9	25	
⋮	⋮	⋮	⋮	C: 9-bit scb. Σ (+2 rows)
8	8	9	25	
–	16	9	25	B: $A+B$
–	16	9	25	
–	–	18	18	C: $C-(A+B)$
–	–	18	18	
–	–	24	24	C: $A\|B+(C-A-B)\|0$
–	–	24	24	
Total #PBS			725	

Remark 3. *In terms of circuit depth, Karatsuba shows to be worth starting from less than 16 bits, provided that a sufficient number of threads is available (cf. Remark 4 in Appendix C) and a careful thread scheduling is applied (similar to that of Table 3). Recall that the threshold for Karatsuba (given in Table 6 in Appendix C) is calculated with respect to the total number of bootstraps, not to the circuit depth. Let us provide an example for 10-bit inputs: we compare the parameters of Karatsuba and schoolbook in Table 4 (more details on 10-bit Karatsuba in Table 7 in Appendix D).*

Table 4: Parameters of 10-bit multiplication. For 8-bit, the depths become equal to 15.

Algorithm	Depth	#PBS	#thr's
Karatsuba	**17**	320	**36**
Schoolbook	19	**280**	100

Although Karatsuba has a lower bootstrapping depth as well as requires a lower number of threads, such an approach requires careful parallelization (as per Table 7), which is currently out of the scope. Therefore, we implement the multiplication threshold t_M *as per Table 6. Also, a little experiment on* *a 12-threaded machine shows that 20-bit multiplication is faster with t_M as proposed (i.e., 10- and 11-bit multiplications via schoolbook), compared to calling Karatsuba starting from 10-bit inputs, achieving 18.0 s and 20.8 s, respectively.*

5.4.2 Experimental Results

We summarize the results of our benchmarks in Table 5, where one can find a performance comparison of Parmesan and Concrete `v0.2`$_\beta$ as well as the analytical quantities for Parmesan.

In Figure 8, we display approximate, per-thread processor load measured during a calculation of the maximum of 32-bit (encrypted) inputs. In that figure, one may spot the expected behavior of operations called within maximum (cf. Algorithm 7); one may observe high loads of:

1. 32 threads, 2 steps $\sim$ 32-bit subtraction (line 1 of that algorithm),
2. 9 threads, 1 step $\sim$ first step of signum's recursion (line 3; calls Algorithm 6 with a 33-bit input),
3. 3 threads, 1 step $\sim$ second step of signum's recursion,
4. 1 thread, 1 step $\sim$ the non-negativity function at the end of signum,
5. 32 threads, 1 step $\sim$ maximum selector (line 5).

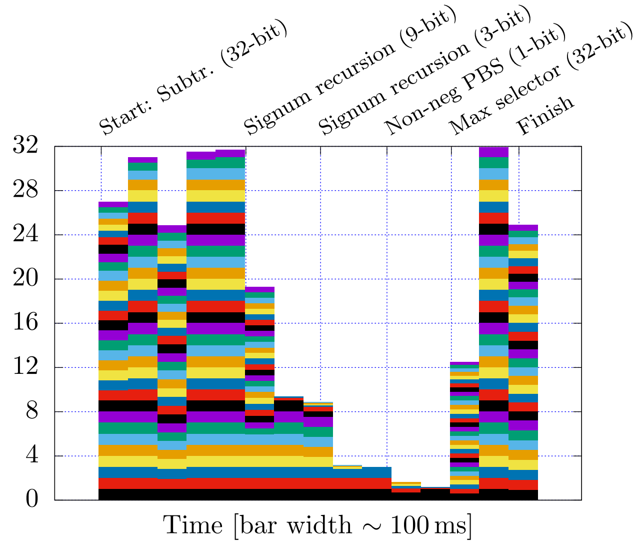


Figure 8: Approximate per-thread processor load measured during a calculation of 32-bit maximum on a 128-threaded supercomputer node. Other than the first 32 threads are omitted due to their negligible load.

Table 5: Benchmarking results of Parmesan vs. Concrete. Experimental machines with a 12-threaded Intel Core i7-7800X processor and with two 64-threaded AMD EPYC 7543 processors were used, in captions as "12-thr." and "128-thr.", respectively. Observed quantities are described in Section 5.4.1, "Speed-Up" gives the speed-up factor of Parmesan over Concrete on respective machine. For Concrete's addition/subtraction, the timings are amortized over three calls, since bootstrapping is only done during the third call; cf. Remark 2.

		Parmesan						**Concrete** `v0.2`$_\beta$		**Speed-Up**	
Operation	$n =$ #bits	Circ. depth	#PBS	Ideal #thr's	Eff. [%]	12-thr. [ms]	128-thr. [ms]	12-thr. [ms]	128-thr. [ms]	12-thr.	128-thr.
PBS	–	–	–	–	–	110	140	–	–	–	–
Add/Sub (Alg. 1, 1; amort. for Concrete)	4	2	$2n$	n	100	220	420	270	480	1.2	1.1
	8					330	400	550	820	1.7	2.1
	16					570	390	1 150	1 310	2.0	3.4
	32					990	450	2 270	2 260	2.3	4.9
Scalar Mul (Alg. 2) #bits $= 16$, val's of $k \to$	4 095	2	$2n$	16	100	580	460	16 340	16 750	28	36
	4 096	0	0	–	–	≈ 0	≈ 0	1 720	1 630	$\approx \infty$	$\approx \infty$
	4 097	2	$2n$	16	100	580	450	1 720	1 600	3.0	3.6
	805	6	114	22	86	1 900	1 350	7 560	7 380	4.0	5.2
	3 195	6	114	22	86	1 890	1 370	10 660	10 390	5.6	7.6
Mul (Alg. 3; mod 2^n in Concrete)	4	7	40	16	36	950	1 490	1 230	2 080	1.3	1.4
	8	15	176	64	18	3 330	3 290	4 600	5 390	1.4	1.6
	16	23	725	81	39	10 990	6 770	18 580	18 620	1.7	2.8
	32	41	2 617	289	22	38 440	15 260	72 160	72 780	1.9	4.8
Squ (Alg. 5; mod 2^n in Concrete)	4	5	24	5	96	650	1 020	1 230	2 020	1.9	2.0
	8	11	122	16	69	2 200	2 430	4 620	5 370	2.1	2.2
	16	19	488	64	40	7 920	5 090	18 560	18 490	2.3	3.5
	32	27	1 837	≤ 129	≥ 53	27 770	10 100	72 200	71 060	2.6	7.0
Signum (Alg. 6)	32	3	11	8	46	390	470	(not implemented)		–	–
Maximum (Alg. 7)	32	6	109	32	57	1 880	1 370				
Rounding (at 5th bit, Alg. 8)	32	4	56	27	52	1 140	1 030				

5.4.3 Discussion

In our benchmarks, we compare two different approaches for basic integer arithmetic over TFHE-encrypted data, implemented in our Parmesan Library and in the Concrete Library, respectively.

Parmesan's arithmetic is based on an algorithm for parallel addition, which uses a redundant integer representation, and in the encrypted domain, it employs a 5-bit message space to hold one bit of an encrypted integer. Parallel addition takes constant time (independent of input length), provided that a sufficient number of threads is available. On top of parallel addition, other arithmetic algorithms are implemented with particular respect to parallelization, including additional optimizations: ASC*s and the window method for scalar multiplication, Karatsuba algorithm for multiplication, divide-and-conquer strategy for squaring, etc.

On the other hand, Concrete's arithmetics employ the standard sequential algorithm for addition. In certain sense, its integer representations are also redundant, as it allows buffering the carry for a limited number of additions, without bootstrapping. However, in the latest beta version of Concrete, there is neither parallelization nor any other optimization of basic arithmetic implemented yet. E.g., for scalar multiplication, this we can easily spot if we compare the results with our particular choices of inputs (cf. Section 5.3.2): namely for $k = 4\,096 = 4^6$, no bootstrapping would be needed in the base-4 representation (our setup of Concrete), however, the timing reveals that many bootstraps occur.

Although there are substantial differences between these two libraries, they are both built upon the same TFHE implementation (that of `concrete-core` crate), and they both use TFHE parameters provided in the `concrete-int` crate (i.e., we may expect the same "quality"). Hence, in certain sense, our benchmarks provide an insight into the direct comparison of these two approaches. E.g., for 16-bit squaring on the 12-threaded processor, Parmesan on the one hand achieves a speed-up by a factor of 2.3, on the other hand, it employs all of the 12 threads, as opposed to Concrete, which runs on a single thread only. However, as stated in Sections 5.1 and 5.2, neither of these libraries is in a production version, therefore, our results shall not be perceived as definitive.

Generally speaking, the redundant representation, required by parallel addition, introduces a certain computational overhead, which can be mitigated by a sufficient number of threads. Therefore, the choice of approach depends on the optimization goal: if minimizing the actual execution time is the priority, we recommend a parallel approach, if minimizing the computational cost (in terms of total CPU time) is the priority, we recommend a sequential approach.

6 Conclusion

We propose and implement parallel algorithms for a fast integer arithmetic over TFHE-encrypted data. We compare our library – the Parmesan Library – with the Concrete Library: for 32-bit inputs on an ordinary 12-threaded server processor, we observe speed-ups by a factor of 2.3 for addition, 1.9 for multiplication, and 2.6 for squaring. On a supercomputer's node with 128 threads, the speed-ups are even higher.

Particular speed-ups are achieved for scalar multiplication, for which we propose a new technique based on the window method and a special kind of addition chains denoted ASC*. E.g., the calculation of $4\,095$ times an encrypted 16-bit integer achieves a $28\times$ speed-up over Concrete on the 12-threaded machine. The advantage of our technique is a combination of multiple factors that our method employs:

- *subtraction*, which goes at the same cost as addition,

- *free doubling* in the radix-based representation, and
- *pre-computed* ASC**s* for up to 12-bit windows.

Besides integer arithmetic, we implement three signum-based operations: signum (also used for number comparison) and maximum, which go with significant optimizations compared to previous work [34], and rounding. These operations aim at completing the most common integer operations.

Thanks to the generic form of our algorithms, the underlying TFHE scheme might be easily replaced with another, LUT-based FHE scheme in the future.

Future Directions & Open Problems

For the Parmesan Library, there are several aspects to be considered:

- finalize the standard computer arithmetic by implementing:
  - bit operations (`&`, `|`, `!`, . . .) and integer division,
  - conversions between the standard and the signed binary representation (if needed, e.g., for bit operations),
  - support for `(u)intXY`-like types, etc.
- algorithms for other common non-atomic operations (like maximum),
- other optimizations taking TFHE batching [13] into account,
- proper thread scheduling (cf. Remark 3) and dedicated optimizations with respect to a fixed number of threads,
- check how the (very recent) *FINAL Scheme* [38] compares to TFHE as the underlying FHE scheme.

From the theoretical point of view, the most interesting open problem is the possible NP-completeness of ASC*s. For the moment, we also do not provide any argument of optimality for our ASC*s, which we generated by a brute-force method for only up to 12-bit integers.

References

[1] Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

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

[3] Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Parameter optimization & larger precision for (t)fhe. 2022.

[4] Daniel J Bernstein. Multidigit multiplication for mathematicians. 2001.

[5] Andrew D Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951.

[6] 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.

[7] 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.

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

[9] 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.

[10] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 34–54. Springer, 2019.

[11] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 360–384. Springer, 2018.

[12] 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.

[13] 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.

[14] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *International Symposium on Cyber Security Cryptography and Machine Learning*, pages 1–19. Springer, 2021.

[15] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved Programmable Bootstrapping with Larger Precision and Efficient Arithmetic Circuits for TFHE. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 670–699. Springer, 2021.

[16] 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.

[17] Sangeeta Chowdhary, Wei Dai, Kim Laine, and Olli Saarikivi. Eva improved: Compiler and extension library for ckks. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 43–55, 2021.

[18] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography, Second Edition.* Chapman & Hall/CRC, 2nd edition, 2012.

[19] Concrete: State-of-the-art TFHE library for boolean and integer arithmetics (v0.2). `https://docs.zama.ai/concrete/`, 2022.

[20] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 546–561, 2020.

[21] Peter Downey, Benton Leong, and Ravi Sethi. Computing sequences with addition chains. *SIAM Journal on Computing*, 10(3):638–646, 1981.

[22] 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.

[23] Robin Geelen, Michiel Van Beirendonck, Hilder VL Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios Dimou, Ingrid Verbauwhede, Frederik Vercauteren, et al. Basalisc: Flexible asynchronous hardware accelerator for fully homomorphic encryption. *arXiv preprint arXiv:2205.14017*, 2022.

[24] 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.

[25] Craig Gentry, Shai Halevi, and Nigel P Smart. Better bootstrapping in fully homomorphic encryption. In *International Workshop on Public Key Cryptography*, pages 1–16. Springer, 2012.

[26] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Annual Cryptology Conference*, pages 75–92. Springer, 2013.

[27] 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.

[28] Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, William Lam, Royce Wilson, Asra Ali, Eric P Astor, Itai Zukerman, Sam Ruth, Christoph Dibak, et al. A general purpose transpiler for fully homomorphic encryption. *arXiv preprint arXiv:2106.07893*, 2021.

[29] Torbjörn Granlund and the GMP development team. GNU MP: The GNU Multiple Precision Arithmetic Library, 2022.

[30] 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.

[31] Marc Joye. Sok: Fully homomorphic encryption over the [discretized] torus. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 661–692, 2022.

[32] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962.

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

[34] Jakub Klemsa and Melek Önen. Parallel Operations over TFHE-Encrypted Multi-Digit Integers. In *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy*, CODASPY '22, page 288–299, New York, NY, USA, 2022. Association for Computing Machinery.

[35] 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.

[36] Kenji Koyama and Yukio Tsuruoka. Speeding up elliptic cryptosystems by using a signed binary window method. In *Annual International Cryptology Conference*, pages 345–357. Springer, 1992.

[37] PARMESAN: Parallel ARithMEticS on tfhe ENcrypted data. `https://github.com/fakub/parmesan`, 2021.

[38] Hilder VL Pereira and Nigel P Smart. Final: Faster fhe instantiated with ntru and lwe. In *Advances in Cryptology–ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part II*, volume 13792, page 188. Springer Nature, 2023.

[39] 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.

[40] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[41] Arnold Schönhage and Volker Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7(3):281–292, 1971.

[42] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. Heco: Automatic code optimizations for efficient fully homomorphic encryption. *arXiv preprint arXiv:2202.01649*, 2022.

[43] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. Sok: Fully homomorphic encryption compilers. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1092–1108. IEEE, 2021.

[44] J Wilson. The Multiply and Fourier Transform Unit: A Micro-Scale Optical Processor, 2022. Accessed: 2022-09-24.

A Conversions Between Binary Representations

Recall that since $\mathcal{A}_2 \subset \bar{\mathcal{A}}_2$, no conversion is needed from the standard to the signed binary representation. Let us discuss the opposite conversion in more detail. First, note that in the signed binary, the leading bit determines the sign of the represented integer. Therefore, we outline this conversion only for positive integers; find it in Algorithm 9.

Algorithm 9 Conversion from the signed to the standard binary representation (assuming a positive input).

Input: $(2, \bar{\mathcal{A}}_2)$-representation $\bar{\mathbf{x}} \in \bar{\mathcal{A}}_2{}^n$ of $X \in \mathbb{N}$, for some $n \in \mathbb{N}$ ($\bar{\mathbf{x}}$ has a positive leading bit),
Output: $(2, \mathcal{A}_2)$-representation $\mathbf{x} \in \mathcal{A}_2{}^n$ of X.

```
1: $\mathbf{x} \leftarrow \bar{\mathbf{x}}$
2: for $i = 0 \ldots n-1$ do
3:     if $x_i < 0$ then
4:         $x_i \leftarrow x_i + 2$
5:         $x_{i+1} \leftarrow x_{i+1} - 1$
6:     end if
7: end for
8: return $\mathbf{x}$
```

For negative integers, we suggest two options:

1. We remember the negative sign, we flip all signs and we proceed with Algorithm 9 normally.

2. We extend the (negative) input by zeros to a pre-determined length $\bar{n} \geq n$ and we run Algorithm 9. This gives us a representation with leading $\bar{1}$ at the $\bar{n}$-th position, which we trim and we obtain $\mathbf{x}' \in \mathcal{A}_2{}^{\bar{n}}$. Such an $\mathbf{x}'$ is a standard complement code of length $\bar{n}$ for $X < 0$. E.g., for an $\bar{n} = 8$-bit representation, we have $(\bar{1}0\bar{1}0\bar{1}0\bullet)_2 \xrightarrow{Alg.\ 9} (\bar{1}11010110\bullet)_2 \sim$ `(int8_t)(0b11'01'01'10)` $= -42$.

Also, recall that for the "opposite" conversion, there does not exist a parallel algorithm. Indeed, the existence of such a parallel algorithm would allow the following scenario: one may convert (trivially) from the standard to the signed binary, perform addition, and convert back, all in parallel. This contradicts the impossibility of parallel addition over a non-redundant integer representation (in this example the standard binary), as shown by Kornerup [35].

B Average Number of Additions in Scalar Multiplication

We evaluate the average number of additions over 12-bit windows in the Koyama-Tsuruoka recoding (recall that the Koyama-Tsuruoka recoding achieves minimal Hamming weight) using:

1. our 12-bit ASC^*s, and

2. the standard double-and-add/sub method.

E.g., for $k = 885$, Koyama-Tsuruoka recoded as $\mathtt{0b100\bar{1}000\bar{1}0\bar{1}\bar{1}}$: the ASC^* is $(1, 7 = -1 + 1 \cdot 2^3, 223 = -1 + 7 \cdot 2^5, 885 = -7 + 223 \cdot 2^2)$, which requires 3 additions, whereas the standard approach

Table 6: Bootstrapping complexity of the schoolbook (Scb.) and the Karatsuba (Kar.) multiplication algorithms with different input bit-lengths (Bits). In Karatsuba, we consider splitting odd numbers such that the longer part is at LSB (cf. Section 3.3.3, item 4).

Bits	...	8	...	14	15	16	17	18	19	...	32	...
Scb.	**...**	**176**	**...**	**560**	**645**	736	**833**	936	1 045	...	3 008	...
Kar.	...	221	...	572	678	**725**	843	**896**	**1 026**	**...**	**2 617**	**...**

gives $(1, 7 = -1 + 1 \cdot 2^3, 111 = -1 + 7 \cdot 2^4, 443 = -1 + 111 \cdot 2^2, 885 = -1 + 443 \cdot 2^1)$, which requires 4 additions.

First, we evaluate the Koyama-Tsuruoka recoding for all odd integers in the interval $[1, 4\,095]$, we trim them to 12 LSBs and we take the absolute value, obtaining 1 792 unique values. Then, we evaluate both ASC^*s and double-and-add/sub, totalling 5 558 and 6 956 additions, respectively. For simplicity, we assume that those 1 792 values are uniformly distributed. This gives us an average of 3.10 additions for ASC^*s and 3.88 additions for double-and-add/sub.

C Threshold for Karatsuba Algorithm

Karatsuba algorithm has a lower asymptotic complexity than the schoolbook algorithm ($O(n^{\log 3})$ vs. $O(n^2)$), however, for short inputs, schoolbook is better. Hence, the aim is to derive threshold t_M, under which the schoolbook algorithm outperforms Karatsuba. In the clear as well as in the encrypted domain, the threshold t_M needs to be evaluated with respect to the complexity of all involved operations in that domain.

Remark 4. *As outlined in Remark 1, it is important to choose a complexity measure. For algorithms like parallel addition/subtraction, we aim at achieving the* lowest bootstrapping depth *– they require as many threads as the number of bits, which makes this choice reasonable with existing (highly) multi-threaded CPUs.*

However, for multiplication, parallelization is enormous in the first couple of steps and then it drops down rapidly. With a limited number of threads, we are closer to the "single-threaded" scenario, where we aim at minimizing the total number of bootstraps*, which we suggest to apply for multiplication.*

Both schoolbook and Karatsuba multiplication can be constructed using just *addition* (cf. Algorithm 1) and *single-bit multiplication* (cf. Algorithm 4), while *addition* requires $A = 2$ bootstraps (per bit), whereas *single-bit multiplication* requires $M = 1$ bootstrap. We evaluate the bootstrapping complexity of the schoolbook algorithm for n-bit inputs as

$$B_{\times}^{(s)}(n) = M \cdot n^2 + A \cdot n \cdot (n-1) = 3n^2 - 2n. \tag{15}$$

We use this result to evaluate the complexity of a first-level Karatsuba, which is sufficient to find the threshold t_M – indeed, recursion emerges for longer inputs, where also the halved input is longer than t_M. We observe that Karatsuba outperforms the schoolbook algorithm starting from around $n = 16$-bit inputs; find the concrete bootstrapping complexities in Table 6. It shows that there is no single threshold t_M, instead, there is a slight overlap.

D Thread Scheduling

We suggest thread scheduling for a potential 10-bit Karatsuba multiplication in Table 7. Next, we suggest thread scheduling for 32-bit multiplication and for 4-, 8-, 16- and 32-bit squarings in Tables 8, 9, 10, 11 and 12, respectively. Note that 16-bit multiplication is covered in Table 3.

Table 7: A suggestion of thread scheduling for the calculation of intermediate values A, B, and C, followed by their aggregation, in potential 10-bit Karatsuba multiplication (intentionally under the threshold t_M to compare with schoolbook), splitting the input into two 5-bit parts. Using 36 threads in 17 steps, totalling 320 bootstraps.

A	B	C	Total #thr's	*Comment*
25	–	5|5	35	C: $\mathbf{r}_1 + \mathbf{r}_0 \mid \mathbf{s}_1 + \mathbf{s}_0$
–	25	5|5	35	
–	–	36	**36**	C: 6-bit pairwise mul.
5	5	6	16	A, B: 5-bit schoolbook summation (8 rows); C: 6-bit scb. Σ (+2 rows)
5	5	6	16	
⋮	⋮	⋮	⋮	
5	5	6	16	
–	10	6	16	B: $A + B$
–	10	6	16	
–	–	12	12	C: $C - (A{+}B)$
–	–	12	12	
–	–	15	15	C: $A \| B + (C{-}A{-}B) \| 0$
–	–	15	15	
Total #PBS			320	

E Algorithm for Squaring of Short Inputs

We provide the full algorithm for squaring of short inputs as Algorithm 10. Note that columns of LUTs in that algorithm hold squares of respective selector in binary: e.g., the 5th column contains 100110, which is a (reversed) binary representation of $5^2 = 25$. The bit at 2^1 position (i.e., y_1) is always zero thanks to the fact $a^2 \bmod 4 \in \{0, 1\}$. Due to line 1, we need $2^{2\Delta} \geq (2^2)^2 + (2^1)^2 + (2^0)^2 = 21$.

Table 8: A suggestion of thread scheduling for the calculation of intermediate values A, B, and C, followed by their aggregation, in 32-bit Karatsuba multiplication, splitting the input into two 16-bit parts. Note that C is calculated via 17-bit schoolbook algorithm; cf. Table 6. Using 289 threads in 41 steps, totalling 2 617 bootstraps.

A	B	C	Total #thr's	*Comment*
–	–	16|16	32	C: $\mathbf{r}_1 + \mathbf{r}_0 \mid \mathbf{s}_1 + \mathbf{s}_0$
–	–	16|16	32	
–	–	289	**289**	C: 17-bit pairwise mul.
80	80	17	177	A, B: 16-bit Karatsuba (23 rows);
80	80	17	177	
⋮	⋮	⋮	⋮	C: 17-bit scb. Σ (+9 rows)
24	24	17	65	
–	33	17	50	B: $A + B$
–	33	17	50	
–	–	17	17	
⋮	⋮	⋮	⋮	
–	–	17	17	
–	–	34	34	C: $C - (A{+}B)$
–	–	34	34	
–	–	35	35	C: $(C{-}A{-}B)\|0 + B$
–	–	35	35	
–	–	33	33	C: $\ldots + A\|0$
–	–	33	33	
Total #PBS			2 617	

Table 9: A suggestion of thread scheduling for the calculation of intermediate values A, B, and C, followed by their aggregation, in 4-bit Divide & Conquer squaring. Using 5 threads in 5 steps, totalling 24 bootstraps.

A	B	C	Total #thr's	*Comment*
–	–	4	4	C: 2-bit pairwise mul.
3	–	2	**5**	A, B: 2-bit squ. (direct);
–	3	2	**5**	C: 2-bit scb. Σ
–	–	5	**5**	C: $A\|B + C\|0$
–	–	5	**5**	
Total #PBS			24	

Table 10: A suggestion of thread scheduling for the calculation of intermediate values A, B, and C, followed by their aggregation, in 8-bit Divide & Conquer squaring. Using 16 threads in 11 steps, totalling 122 bootstraps.

A	B	C	Total #thr's	*Comment*
–	–	16	**16**	C: 4-bit pairwise mul.
4	4	4	12	
5	5	4	14	A, B: 4-bit D'n'Q (5 rows);
⋮	⋮	⋮	⋮	C: 4-bit scb. Σ (+1 row)
5	5	4	14	
–	–	4	4	
–	–	8	8	C: $C\|0 + B$
–	–	8	8	
–	–	9	9	C: $\ldots + A\|0$
–	–	9	9	
Total #PBS			122	

Table 11: A suggestion of thread scheduling for the calculation of intermediate values A, B, and C, followed by their aggregation, in 16-bit Divide & Conquer squaring. Using 64 threads in 19 steps, totalling 488 bootstraps.

A	B	C	Total #thr's	*Comment*
–	–	64	**64**	C: 8-bit pairwise mul.
16	16	8	40	
⋮	⋮	⋮	⋮	A, B: 8-bit D'n'Q (11 rows);
9	9	8	26	
–	–	8	8	C: 8-bit scb. Σ
–	–	8	8	
–	–	8	8	
–	–	16	16	C: $C\|0 + B$
–	–	16	16	
–	–	18	18	C: $\ldots + A\|0$
–	–	18	18	
Total #PBS			488	

Table 12: A suggestion of thread scheduling for the calculation of intermediate values A, B, and C, followed by their aggregation, in 32-bit Divide & Conquer squaring. Using 129 threads in 27 steps, totalling 1 837 bootstraps.

A	B	C	Total #thr's	*Comment*
–	–	80	80	
–	–	80	80	
–	–	81	81	A, B: 16-bit D'n'Q
–	64	25	89	(19 rows);
64	40	25	**129**	C: 16-bit Karatsuba
⋮	⋮	⋮	⋮	(23 rows)
18	18	24	60	
18	–	24	42	
–	–	33	33	C: $C\|0 + B$
–	–	33	33	
–	–	35	35	C: $\ldots + A\|0$
–	–	35	35	
Total #PBS			1 837	

Algorithm 10 Short-Input Squaring.

Input: 3-bit representation $(x_2x_1x_0\bullet)_2 \in (\bar{\mathcal{A}}_2)^3$ of $X \in \mathbb{Z}$,
Output: 6-bit representation $(y_5 \ldots y_0\bullet)_2 \in (\bar{\mathcal{A}}_2)^6$ of X^2.

```
1: $X \leftarrow \mathsf{eval}_2(x_2x_1x_0\bullet)$                          ▷ no bootstrap needed; cf. (9)
2: in parallel do
3:     $y_0 \leftarrow (0,1,0,1,0,1,0,1\| 1,0,1,0,1,0,1)[X]$
4:     $y_1 \leftarrow 0$                                                       ▷ always 0
5:     $y_2 \leftarrow (0,0,1,0,0,0,1,0\| 0,1,0,0,0,1,0)[X]$
6:     $y_3 \leftarrow (0,0,0,1,0,1,0,0\| 0,0,1,0,1,0,0)[X]$
7:     $y_4 \leftarrow (0,0,0,0,1,1,0,1\| 1,0,1,1,0,0,0)[X]$
8:     $y_5 \leftarrow (0,0,0,0,0,0,1,1\| 1,1,0,0,0,0,0)[X]$
9: end parallel
10: return $(y_5 \ldots y_0\bullet)_2$
```