Linear Depth Integer-wise Homomorphic Division

Hiroki Okada[1], Carlos Cid[2], Hidano Seira[1], and Shinsaku Kiyomoto[1]

[1] KDDI Research, Inc.
Saitama, Japan
[2] Royal Holloway, University of London
Egham, Surrey, United Kingdom
ir-okada@kddi-research.jp

Abstract. We propose a secure *integer-wise* homomorphic division algorithm on fully homomorphic encryption schemes (FHE). For integer-wise algorithms, we encrypt plaintexts as integers without encoding them into bit values, while in *bit-wise* algorithms, plaintexts are encoded into binary and bit values are encrypted one by one. All the publicly available division algorithms are constructed in bit-wise style, and to the best of our knowledge there are no known integer-wise algorithm for secure division. We derive some empirical results on the FHE library HElib and show that our algorithm is 2.45x faster than the fastest bit-wise algorithm. We also show that the multiplicative depth of our algorithm is $O(l)$, where l is the integer bit length, while that of existing division algorithms is $O(l^2)$. Furthermore, we generalise our secure division algorithm and propose a method for secure calculation of a general 2-variable function. The order of multiplicative depth of the algorithm, which is a main factor of the complexity of a FHE algorithm, is exactly the same as our secure division algorithm.

Keywords: Fully homomorphic encryption, HElib, Secure integer arithmetic, Circuit depth

1 Introduction

Fully Homomorphic Encryption. A fully homomorphic encryption (FHE) scheme presents a way to perform arbitrary calculations on encrypted data without the requirement of decryption. The first construction of FHE [11, 12] was given by Gentry in 2009. Several improvements [4–6, 10, 13, 14, 25, 27] have followed since then, developing a diversity of features and complexity assumptions. HElib [17–19] is a library for FHE widely used in applications, which implements the BGV scheme [3]. It allows for the "packing" of ciphertexts and single instruction multiple data (SIMD) computations, amortizing the cost for certain tasks.

There are numerous applications of FHE, but one of the most remarkable is privacy-preserving delegated computations, such as privacy-preserving machine learning as a service. In the service, users do not wish to reveal their sensitive data to the server, and the server does not want to reveal the cognitive model

to users. FHE enables these scenarios in an elegant way with non-interactivity. However, because of the inefficiency of existing FHE schemes, most applications are constructed evading the *non-crypto-friendly* calculations such as comparison, division, and some non-linear functions.

Bit-wise encryption vs. Integer-wise encryption. Most FHE schemes, including the BGV scheme, feature *integer-wise* and *bit-wise* encryption; the size of the plaintext space in the scheme is variable. In bit-wise encryption the plaintext space is $\mathbb{Z}_2$, and in integer-wise encryption the plaintext space is $\mathbb{Z}_p$ where $p > 2$. Because the main libraries of FHE, including HElib, do not support some basic integer arithmetic operations such as division and comparison, several studies [8,9,28] have been performed on improved arithmetic. The proposed algorithms are primarily for bit-wise encryption, and they simply leverage the existing algorithms for such operations on the bit-wise circuit such as Ripple carry adder, long multiplication, and non-restoring division. Although bit-wise encryption can perform integer comparison very efficiently [7], integer addition and multiplication are not practical since they require homomorphic multiplication. On the other hand, integer-wise arithmetic can naturally perform integer addition and multiplication efficiently, and recent remarkable privacy-preserving machine learning such as [1, 2, 15, 20] use integer-wise encryption. However, the algorithm for these applications evades arithmetic such as division and comparison, which are believed to be inefficient. While a concrete algorithm for secure integer-wise comparison has been recently proposed in [22], to the best of our knowledge there is no known concrete algorithm for secure integer-wise division algorithm. Emergence of efficient algorithms for basic arithmetic operations such as division and comparison will undoubtedly increase options to optimise higher level applications of secure computation.

Our contribution. We present a new concrete algorithm for privacy preserving *integer-wise* division. Although several studies have been performed on privacy preserving *bit-wise* division algorithms [8, 9, 28], there is no known concrete algorithm for the *integer-wise* version. We implement our division algorithm using HElib, and test its performance. The experimental results show that our algorithm performs 2.45 times faster than the fastest bit-wise algorithm [8]. We also theoretically analyse the multiplicative depth, which is a barometer for the complexity for the FHE-based algorithm. While the order of the multiplicative depth of existing division algorithms [8, 9, 28] is $O(l^2)$ for l-bit size integers, we show that our algorithm can perform with $O(l)$.

Furthermore, we generalise our secure division algorithm and propose an algorithm for secure calculation of a *general 2-variable function*; the order of multiplicative depth of the algorithm is $O(l)$, which is the same as our secure division algorithm. This is the first result to construct a concrete algorithm for performing the general 2-variable function, expanding the FHE application diversity.

2 Preliminaries

2.1 Notation

In the FHE construction, a ring R is used, whose elements are written in lower case; for example, $r \in R$. For an integer q, we use R_q to denote R/qR. For $a \in R$, we use the notation $[a]_q$ to refer to a mod q, with coefficients reduced to the range $(-\frac{q}{2}, \frac{q}{2}]$. A concrete instantiation for our applications is the quotient polynomial ring $R_q = \mathbb{Z}_q[x]/\Phi_m(x)$, where q is a prime and $\Phi_m(x)$ is the m-th cyclotomic polynomial. We denote a ciphertext of r, which is encrypted with a FHE scheme, by C_r.

We denote the logarithm to base 2 and the natural logarithm as $\log(\cdot)$ and $\ln(\cdot)$, respectively. We denote vectors in bold. The notation $\boldsymbol{v}[i]$ refers to the i^{th} coefficient of $\boldsymbol{v}$, while the scalar product of two vectors $\boldsymbol{u}, \boldsymbol{v} \in R^n$ is denoted as $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \sum_{i=1}^{n} \boldsymbol{u}[i] \cdot \boldsymbol{v}[i] \in R$. By $(\boldsymbol{a} \| \boldsymbol{b})$ we denote the concatenation of two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$. We write $\boldsymbol{s} \xleftarrow{U} \mathcal{S}$ to denote the process of sampling $\boldsymbol{s}$ uniformly at random over $\mathcal{S}$; when the set $\mathcal{S}$ is clear from the context, we will write $e \leftarrow \chi$ to denote the process of sampling e according to the probability distribution χ over $\mathcal{S}$.

2.2 The BGV scheme

A FHE scheme is a public-key cryptographic scheme that includes two operations $(+, \cdot)$ on ciphertexts such that: $\mathsf{Dec}(C_a + C_b) = a + b$, $\mathsf{Dec}(C_a \cdot C_b) = a \cdot b$. The BGV scheme [3] is a widely used FHE scheme for practical applications, which is implemented in the FHE library HElib [16]. The security of the scheme is based on the standard assumptions of the Learning with Error (LWE) problem [23] or Ring-LWE (RLWE) problem [21]. This is in contrast to the earlier FHE constructions [11, 12] which were based on ad-hoc average-case assumptions on ideal lattice problems.

The Basic Encryption Scheme. BGV is a public-key cryptography scheme $\mathsf{E} = (\mathsf{E.Setup}, \mathsf{E.SecretKeyGen}, \mathsf{E.PublicKeyGen}, \mathsf{E.Enc}, \mathsf{E.Dec})$ defined as follows.

- $\mathsf{Setup}(1^\lambda)$. Given the security parameter λ as input, set an integer $m = m(\lambda)$ that defines the cyclotomic polynomial $\Phi_m(x)$, and the odd modulus $q = q(\lambda)$. If $R = \mathbb{Z}[x]/\Phi_m(x)$, the underlying working ring is $R_q = \mathbb{Z}_q[x]/\Phi_m(x)$. Set a plaintext modulus p that is relatively prime to q, with the plaintext space given by $R_p = \mathbb{Z}_p[x]/\Phi_m(x)$. Set a noise distribution $\chi = \chi(\lambda)$ over the underlying working ring, and $N = N(\lambda) = \text{polylog}(q)$. Output $params = (R, m, q, p, \chi, N)$.
- $\mathsf{SecretKeyGen}(params)$. Sample $s \leftarrow \chi$. Output the secret key $sk = \boldsymbol{s} := (1, s) \in R_q^2$.
- $\mathsf{PublicKeyGen}(params, sk)$. Take as input the secret key $sk = \boldsymbol{s} = (1, s)$ and $params$. Sample $\boldsymbol{a} \xleftarrow{U} R_q^N$ and $\boldsymbol{e} \leftarrow \chi^N$. Set $\boldsymbol{b} := s\boldsymbol{a} + p\boldsymbol{e} \in R_q^N$, and output the public key defined as $pk = \boldsymbol{A} := (\boldsymbol{b}, -\boldsymbol{a}) \in R_q^{N \times 2}$. Notice that $\boldsymbol{A} \cdot \boldsymbol{s} = \boldsymbol{b} - s\boldsymbol{a} = p\boldsymbol{e}$, from the definition of $\boldsymbol{b}$.

- $\mathsf{Enc}(params, pk, m)$. To encrypt a message $m \in R_p$, set $\boldsymbol{m} = (m, 0) \in R_p^2$, sample $\boldsymbol{r} \xleftarrow{U} R_p^N$ and output the ciphertext $\boldsymbol{c} := \boldsymbol{m} + \boldsymbol{r}^\top \boldsymbol{A} \in R_q^2$.
- $\mathsf{Dec}(params, sk, \boldsymbol{c})$. Output the message $m := [[\langle \boldsymbol{c}, \boldsymbol{s} \rangle]_q]_p$.

The decryption works because

$$[[\langle \boldsymbol{c}, \boldsymbol{s} \rangle]_q]_p = [[(\boldsymbol{m} + \boldsymbol{r}^\top \boldsymbol{A}) \cdot \boldsymbol{s}]_q]_p = [[m + p\boldsymbol{r}^\top \boldsymbol{e}]_q]_p = [m + p\boldsymbol{r}^\top \boldsymbol{e}]_p = m,$$

where the third equality holds since $\boldsymbol{r}$ and $\boldsymbol{e}$ have small enough entries so that the value $m + p\boldsymbol{r}^\top \boldsymbol{e}$ is smaller than the modulus q.

The FHE Scheme. As a FHE scheme, the BGV scheme supports addition and multiplication over the plaintext and ciphertext spaces. Let $\boldsymbol{c}_a$ and $\boldsymbol{c}_b$ be ciphertexts of plaintexts a and b under the same key sk, respectively. The addition of two ciphertexts is simply a component-wise addition, i.e.

$$\boldsymbol{c}_a + \boldsymbol{c}_b = (\boldsymbol{c}_a[0], \boldsymbol{c}_a[1]) + (\boldsymbol{c}_b[0], \boldsymbol{c}_b[1]) = (\boldsymbol{c}_a[0] + \boldsymbol{c}_b[0], \boldsymbol{c}_b[1] + \boldsymbol{c}_a[1]) = \boldsymbol{c}_{a+b},$$

which is a ciphertext of $a+b \in R_p$. The homomorphic multiplication is performed by the tensor product of two ciphertexts. The tensor product of ciphertexts

$$\boldsymbol{c}_{a \cdot b} := \boldsymbol{c}_a \otimes \boldsymbol{c}_b := (\boldsymbol{c}_a[0]\boldsymbol{c}_b[0], \boldsymbol{c}_a[0]\boldsymbol{c}_b[1] + \boldsymbol{c}_a[1]\boldsymbol{c}_b[0], \boldsymbol{c}_a[1]\boldsymbol{c}_b[1]) \tag{1}$$

is a ciphertext of $a \cdot b \in R_p$ under the new secret key $\boldsymbol{s}' := \boldsymbol{s} \otimes \boldsymbol{s}$. In this way, the homomorphic multiplication increases the size of ciphertexts exponentially. In order to deal with this expanding ciphertext, the BGV scheme features *key switching*. The key switching function $\mathsf{SwitchKey}(\tau_{\boldsymbol{s}' \to \overline{\boldsymbol{s}}}, \boldsymbol{c}', q)$ takes the ciphertext $\boldsymbol{c}'$ under $\boldsymbol{s}'$ and outputs a new ciphertext $\overline{\boldsymbol{c}}$ that encrypts the same message under the secret new key $\overline{\boldsymbol{s}}$. Using this function, we can reduce the size of ciphertext $\boldsymbol{c}_{a \cdot b} \in R_q^3$ to $\overline{\boldsymbol{c}}_{a \cdot b} \in R_q^2 \leftarrow \mathsf{SwitchKey}(\tau_{\boldsymbol{s}' \to \overline{\boldsymbol{s}}}, \boldsymbol{c}_{a \cdot b})$. The BGV scheme also features *modulus switching* techniques, which reduce the magnitude of the noise of the ciphertext by switching the modulus from q to the smaller modulus q'. The modulus switching function $\mathsf{Scale}(\boldsymbol{c}, q, q', p)$, takes a ciphertext $\boldsymbol{c}$ for modulus q and outputs a ciphertext under same secret for modulus q'.

We now briefly describe the BGV FHE scheme. The scheme is a *levelled* FHE scheme.

- $\mathsf{FHE.Setup}(1^\lambda, 1^L)$. Takes as input the security parameter λ and a number of levels L. Set an integer $m = m(\lambda, L)$ that defines the cyclotomic polynomial $\Phi_m(x)$. Let $\mu = \mu(\lambda, L, b) = \theta(\log \lambda + \log L)$ be a parameter to define the bit size of the moduli. For $j = L$ (input level of circuit) to 0 (output level), run $params_j \leftarrow \mathsf{E.Setup}(1^\lambda, 1^{(j+1)\cdot\mu}, b)$ to obtain a list of parameters, including a list of moduli $\{q_L$ ($(L+1) \cdot \mu$ bits), $q_{L-1}, \ldots, q_0$ (μ bits)$\}$.
- $\mathsf{FHE.KeyGen}(params_j)$. For $j = L$ down to 0, do the following:
  1. Run the basic schemes $\boldsymbol{s}_j \leftarrow \mathsf{E.SecretKeyGen}(params_j)$, and $\boldsymbol{A}_j \leftarrow \mathsf{E.PublicKeyGen}(params_j, \boldsymbol{s}_j)$.
  2. Set $\boldsymbol{s}'_j \leftarrow \boldsymbol{s}_j \otimes \boldsymbol{s}_j$.

3. Run $\tau_{\boldsymbol{s}'_j \to \boldsymbol{s}_{j-1}} \leftarrow \mathsf{SwitchKeyGen}(\boldsymbol{s}'_j, \boldsymbol{s}_{j-1})$, where $\mathsf{SwitchKeyGen}$ is a generator function of auxiliary information $\tau_{\boldsymbol{s}'_j \to \boldsymbol{s}_{j-1}}$ that will be used for $\mathsf{SwitchKey}$. (Note that we omit this step when $j = 0$.)
4. Output $sk := \{\boldsymbol{s}_j\}_{j=0}^{L}$, $pk = \{\boldsymbol{A}_j\}_{j=0}^{L}$.

– $\mathsf{FHE.Enc}(params, pk, m)$. Take a message m in R_p. Run $\boldsymbol{c} \leftarrow \mathsf{E.Enc}(params_L, \boldsymbol{A}_L, m)$ of the basic scheme.
– $\mathsf{FHE.Dec}(params, sk, \boldsymbol{c})$. Suppose that the input ciphertext $\boldsymbol{c}$ is under key $\boldsymbol{s}_j$. Here, we know the level (the index j) of the ciphertext from its augmented information. Run $\mathsf{E.Dec}(params_j, \boldsymbol{s}_j, \boldsymbol{c})$.
– $\mathsf{FHE.Eval}(pk, f, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_l)$. Take as input a circuit f for ciphertexts $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_l$. It is assumed that f is a levelled circuit composed of layers of alternating addition and multiplication gates. $\mathsf{FHE.Eval}$ will invoke $\mathsf{FHE.Add}$ and $\mathsf{FHE.Mult}$, which is described next, to compute the circuit. The ciphertext refreshing procedure $\mathsf{FHE.Refresh}$ (described later) is invoked after every multiplication layer, in order to reduce the noise in the ciphertexts and move it to a different level.
  - $\mathsf{FHE.Add}(pk, \boldsymbol{c}_1, \boldsymbol{c}_2)$. Takes two ciphertexts encrypted under the same $\boldsymbol{s}_j$. If they are not under the same key, use $\mathsf{FHE.Refresh}$ to make one of them, the level of which is higher than the other, to be encrypted under $\boldsymbol{s}_j$. Output $\boldsymbol{c}_3 \leftarrow \boldsymbol{c}_1 + \boldsymbol{c}_2 \bmod q_j$.
  - $\mathsf{FHE.Mult}(pk, \boldsymbol{c}_1, \boldsymbol{c}_2)$. Takes two ciphertexts encrypted under the same $\boldsymbol{s}_j$. If they are not under the same key, use $\mathsf{FHE.Refresh}$ to make one of them, the level of which is higher than the other, to be encrypted under $\boldsymbol{s}_j$. Multiply the two ciphertexts, then obtain the new ciphertext $\boldsymbol{c}_3$ under the long secret key $\boldsymbol{s}'_j = \boldsymbol{s}_j \otimes \boldsymbol{s}_j$. $\boldsymbol{c}_3$ is the coefficient vector of $\langle \boldsymbol{c} \otimes \boldsymbol{c}, \boldsymbol{x} \otimes \boldsymbol{x} \rangle$. Then, output $\boldsymbol{c}_4 \leftarrow \mathsf{FHE.Refresh}(\boldsymbol{c}_3, \tau_{\boldsymbol{s}'_j \to \boldsymbol{s}_{j-1}}, q_j, q_{j-1})$.
– $\mathsf{FHE.Refresh}(\boldsymbol{c}, \tau_{\boldsymbol{s}'_j \to \boldsymbol{s}_{j-1}}, q_j, q_{j-1})$. Takes a ciphertext encrypted under $\boldsymbol{s}'_j$, the auxiliary information $\tau_{\boldsymbol{s}'_j \to \boldsymbol{s}_{j-1}}$ for key switching, and the current and next moduli q_j and q_{j-1}. Perform the following.
  1. (Key switching.) Set $\boldsymbol{c}_1 \leftarrow \mathsf{SwitchKey}(\tau_{\boldsymbol{s}'_j \to \boldsymbol{s}_{j-1}}, \boldsymbol{c}, q_j)$, a ciphertext under the key $\boldsymbol{s}_{j-1}$ for modulus q_j.
  2. (Moduli switching.) Set $\boldsymbol{c}_2 \leftarrow \mathsf{Scale}(\boldsymbol{c}_1, q_j, q_{j-1}, p)$, a ciphertext under the key $\boldsymbol{s}_{j-1}$ for modulus q_{j-1}. Output $\boldsymbol{c}_2$.

Multiplicative Depth and Level Parameter L. As mentioned in [3], we do not need to perform $\mathsf{FHE.Refresh}$ after addition. We do not perform $\mathsf{SwitchKey}$ after addition either, since addition does not increase the size of the ciphertext. Moreover, since addition increases the noise much more slowly than multiplication, we do not need to perform Scale after addition either. Finally we also note that in HElib, $\mathsf{FHE.Refresh}$ is performed only after $\mathsf{FHE.Mult}$ is performed.

The parameter L, which indicates the number of levels of arithmetic circuit that the scheme is capable of evaluating, is very important when we estimate the complexity of the FHE circuit. Every time we perform $\mathsf{FHE.Mult}$, we perform $\mathsf{SwitchKey}$ and move the index j to $j - 1$. Thus, basically, we set the level

Table 1: Basic interfaces for homomorphic evaluation in HElib.

HElib Interface	Abbreviation we use
`Ctxt::addCtxt`	$\mathsf{FHE.Add}(\boldsymbol{c}, \boldsymbol{c}')$
`Ctxt::multiplyBy`	$\mathsf{FHE.Mult}(\boldsymbol{c}, \boldsymbol{c}')$
`Ctxt::addConstant`	$\mathsf{FHE.addConst}(\boldsymbol{c}, m')$
`Ctxt::multByConstant`	$\mathsf{FHE.multConst}(\boldsymbol{c}, m')$

parameter L according to the multiplicative depth of the circuit. The level L is related with the complexity of FHE.Add or FHE.Add. Brakerski *et al.* [3] showed the order of complexity is $O(\lambda L^3)$.

2.3 HElib

HElib [16] is a software library that implements the BGV scheme in C++. HElib is based on the number theory library NTL [24]. In addition to the basic scheme, HElib also supports the SIMD feature proposed by Smart and Vercauteren [26]. The SIMD feature enables packing multiple plaintexts into a single element of R_p with the Chinese Remainder Theorem; it also enables parallel component-wise evaluation of the plaintexts in the SIMD "slots". It produces a much better amortised performance, due to parallelisation.

HElib has an interface for the "constant" evaluation, where addition or multiplication by a plaintext is performed for ciphertext. Note that Table 1 shows basic interfaces for homomorphic evaluation in the HElib. These constant evaluations are efficient when the addend or multiplier are not encrypted values. In particular, constant multiplication is quite efficient compared to the homomorphic multiplication; the constant multiplication does not increase the dimension of the ciphertext, and we do not need to perform SwitchKey after the constant multiplication.

2.4 Polynomial interpolation and integer-wise secure comparison

The first integer-wise homomorphic comparison algorithm was proposed in [22]. We refer to the algorithm as Algorithm 1, which is based on the polynomial interpolation technique.

Polynomial interpolation is a process of constructing a polynomial $f(x)$ of degree at most n which satisfies $y_i = f(x_i), i \in \{0, 1, \ldots, n\}$, where $\{x_i, y_i\}$ is given for $n+1$ data points such that $x_i \neq x_j$ when $i \neq j$. We can calculate the polynomial $f(x)$ by

$$f(x) = \sum_{i=0}^{n} \left(\prod_{0 \leq j \leq n, j \neq i} \frac{x - x_j}{x_i - x_j} \right) y_i. \tag{2}$$

Algorithm 1 $\mathsf{Comp}(C_a, C_b)$: Integer-wise Homomorphic Comparison [22]

Input: Ciphertexts C_a, C_b.

Output: $C_{(a \geq b)}$, such that $C_{(a \geq b)} = \begin{cases} C_0 & (a \geq b), \\ C_1 & (a < b). \end{cases}$

1: (Precomputation): Using the polynomial interpolation algorithm, find polynomial $f(x) \in \mathbb{Z}_p[x]$ that satisfies

$$f(x) = \begin{cases} 0 & (x = 0, 1, 2, \ldots, \lfloor \frac{p}{2} \rfloor), \\ 1 & (x = -\lfloor \frac{p}{2} \rfloor, \ldots, -2, -1). \end{cases}$$

For example, with plaintext modulus $p = 7$, the interpolation polynomial $f(x)$ of degree 6 is calculated such that

$$f(-3) = 1, f(-2) = 1, f(-1) = 1, f(0) = 0, f(1) = 0, f(2) = 0, f(3) = 0.$$

For this example, $f(x) = 4x^6 - x^5 - 6x^3 - 4x \in \mathbb{Z}_7[x]$.

2: Homomorphically computes $C_{a-b} = C_a - C_b$.

3: Calculate and output $C_{(a \geq b)} = f(C_{a-b})$.

Note that in Algorithm 1, the polynomial intepolation technique is used to construct the Heaviside step function (i.e, comparison with 0). Our sub-algorithms $\mathsf{ConstDiv}$ and $\mathsf{ConstEq}$ are also constructed based on the polynomial interpolation, as discussed in the next section.

3 Our Algorithm for Integer-wise Homomorphic Division

In this section we present our integer-wise secure division algorithm. In the algorithm, the polynomial interpolation technique is used as a precomputation, similar to the integer-wise comparison algorithm from [22] (Algorithm 1).

In the following, we present in Section 3.1 the overview of the algorithms employed in our secure homomorphic division. We describe the algorithms in detail in Section 3.2. Finally, we analyse their complexity and provide empirical results in Section 3.3.

3.1 Overview

Our integer-wise homomorphic division algorithm $\mathsf{Div}(C_a, C_d)$ is given in Algorithm 2, and is constructed based on the following subroutines:

- (Algorithm 3) $\mathsf{Pows}(C_a)$: Computes powers of a ciphertext C_a.
- (Algorithm 4) $\mathsf{ConstDiv}(C_a^{\mathsf{pow}}, y)$: Integer-wise division by public divisor y.
- (Algorithm 5) $\mathsf{ConstEq}(C_d^{\mathsf{pow}}, y)$: Integer-wise equality check with a public input y.

$\mathsf{ConstDiv}(C_a^{\mathsf{pow}}, y)$ homomorphically computes the ciphertext of a quotient $\lfloor a/y \rfloor$ denoted as $C_{\lfloor a/y \rfloor}$, from the ciphertext of $a \in \mathbb{Z}_p$ denoted as C_a, and public

Algorithm 2 $\mathsf{Div}(C_a, C_d)$: Integer-wise Homomorphic Division

Input: Ciphertexts C_a, C_d.
Output: $C_{\lfloor a/d \rfloor}$
1: $C_{\mathsf{sum}} = 0$
2: $\boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}} = \mathsf{Pows}(C_a)$ ▷ $\boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}} := \{C_a, C_a^2, C_a^3, \ldots, C_a^{p-1}\}$
3: $\boldsymbol{C}_{\boldsymbol{d}}^{\mathsf{pow}} = \mathsf{Pows}(C_d)$ ▷ $\boldsymbol{C}_{\boldsymbol{d}}^{\mathsf{pow}} := \{C_d, C_d^2, C_d^3, \ldots, C_d^{p-1}\}$
4: **for** $i = 0$ to $(p-1)$ **do**
5: $\quad C_{\lfloor a/i \rfloor} = \mathsf{ConstDiv}(\boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}}, i)$
6: $\quad C_{(d=i)} = \mathsf{ConstEq}(\boldsymbol{C}_{\boldsymbol{d}}^{\mathsf{pow}}, i)$
7: $\quad C_{\mathsf{sum}} = C_{\mathsf{sum}} + \mathsf{FHE.Mult}(C_{\lfloor a/i \rfloor}, C_{(d=i)})$ ▷ $\mathsf{FHE.Mult}(C_{\lfloor a/i \rfloor}, C_{(i=d)}) =$
8: **end for** ▷ $C_{\lfloor a/d \rfloor}$ if $i = d$; C_0 otherwise
9: Output $C_{\lfloor a/d \rfloor} = C_{\mathsf{sum}}$

divisor $y \in \mathbb{Z}_p$. $\mathsf{ConstEq}(C_d^{\mathsf{pow}}, y)$ homomorphically computes the ciphertext of the Boolean value $(y = d)$ denoted as $C_{(y=d)}$, where

$$C_{(y=d)} := \begin{cases} C_1 & \text{if } y = d; \\ C_0 & \text{otherwise.} \end{cases}$$

Note that given C_a, C_d for unknown $a, d \in \mathbb{Z}_p$, then for an arbitrary public value $y \in \mathbb{Z}_p$, we have

$$C_{\lfloor a/y \rfloor} * C_{(y=d)} = \begin{cases} C_{\lfloor a/d \rfloor} & \text{if } y = d; \\ C_0 & \text{otherwise.} \end{cases}$$

Our main algorithm for homomorphic division of a by d is then based on a simple idea: calculate $C_{\lfloor a/y \rfloor} * C_{(y=d)}$ for all public values y and sum them:

$$C_{\mathsf{sum}} := \sum_{y \in \mathbb{Z}_p} C_{\lfloor a/y \rfloor} * C_{(y=d)} = C_{\lfloor a/d \rfloor}.$$

3.2 Algorithms

Main algorithm. Algorithm 2 shows the integer-wise secure division algorithm $\mathsf{Div}(C_a, C_d)$. The algorithm takes two ciphertexts C_a and C_d, which are ciphertexts of a and d, respectively, then homomorphically calculates a ciphertext $C_{\lfloor a/d \rfloor}$, the decryption of which gives $\lfloor a/d \rfloor$.

The algorithm first calls Pows for both inputs C_a and C_d, to obtain the list of powers $\boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}}$ and $\boldsymbol{C}_{\boldsymbol{d}}^{\mathsf{pow}}$. This part performs a high number of homomorphic multiplications, but we store these values and can reuse them. Next, in the `for` loop, for all $i \in \{0, 1, 2, \ldots, p-1\}$, we exhaustively calculate $C_{\lfloor a/i \rfloor}$ and $C_{(d=i)}$ with $\mathsf{ConstDiv}$ and $\mathsf{ConstEq}$, then perform the homomorphic multiplication of $C_{\lfloor a/i \rfloor} * C_{(i=d)} := \mathsf{FHE.Mult}(C_{\lfloor a/i \rfloor}, C_{(i=d)})$. Recall that $C_{\lfloor a/i \rfloor} * C_{(i=d)}$ equals to $C_{\lfloor a/d \rfloor}$ if $y = d$, and C_0 otherwise. Thus, at the end of the algorithm we obtain $C_{\lfloor a/d \rfloor} = C_{\mathsf{sum}}$.

Algorithm 3 $\mathsf{Pows}(C_a)$: Computation of powers of C_a

Input: Ciphertexts C_a
Output: $\boldsymbol{C_a^{\mathsf{pow}}} := (C_a, C_a^2, C_a^3, \ldots, C_a^{p-1})$
1: Let $l := \lfloor \log p \rfloor$.
2: **for** $i = 0$ to $(l-1)$ **do**
3: **for** $j = 1$ to 2^i **do**
4: Calculate $C_a^{(2^i+j)} := \mathsf{FHE.Mult}(C_a^{(2^i)}, C_a^j)$.
5: **end for**
6: **end for** ▷ Here, we have $C_a, C_a^2, C_a^3 \ldots, C_a^{2^l}$.
7: **if** $2^l < p-1$ **then**
8: **for** $j = 1$ to $p - 1 - 2^l$ **do**
9: Calculate $C_a^{(2^l+j)} := \mathsf{FHE.Mult}(C_a^{(2^l)}, C_a^j)$.
10: **end for**
11: **end if** ▷ Here, we obtain $C_a^{2^l+1}, \ldots, C_a^{p-1}$.
12: Output $\boldsymbol{C_a^{\mathsf{pow}}} := (C_a, C_a^2, C_a^3, \ldots, C_a^{p-1})$.

$\mathsf{Pows}(\boldsymbol{C_a})$. Algorithm 3 shows the sub-algorithm $\mathsf{Pows}(C_a)$. The algorithm takes a ciphertext C_a as an input and homomorphically calculates the powers of C_a, returning the list of powers $\boldsymbol{C_a^{\mathsf{pow}}} := (C_a, C_a^2, C_a^3, \ldots, C_a^{p-1})$. This sub-algorithm is the most complex part of the main division algorithm Div. In Div, this algorithm is called only once per ciphertext. Note that the multiplicative depth of $\mathsf{Pows}(C_a)$ is $\lceil \log(p-1) \rceil$. For example, when $p = 17$, since $C_a^2 = \mathsf{FHE.Mult}(C_a, C_a)$, $C_a^4 = \mathsf{FHE.Mult}(C_a^2, C_a^2)$, $C_a^8 = \mathsf{FHE.Mult}(C_a^4, C_a^4)$, and $C_a^{16} = \mathsf{FHE.Mult}(C_a^8, C_a^8)$, multiplicative depth is $\log(16) = 4$. See Section 3.3 for a more detailed complexity analysis.

$\mathsf{ConstDiv}(\boldsymbol{C_a^{\mathsf{pow}}}, \boldsymbol{d})$. Algorithm 4 shows the sub-algorithm $\mathsf{ConstDiv}(C_a^{\mathsf{pow}}, d)$. The algorithm takes a list of powers of ciphertext $\boldsymbol{C_a^{\mathsf{pow}}} := (C_a, C_a^2, C_a^3, \ldots, C_a^{p-1})$ and a plaintext divisor d as input, then homomorphically calculates a ciphertext $C_{\lfloor a/d \rfloor}$, the decryption of which gives $\lfloor a/d \rfloor$. Note that this algorithm does not perform homomorphic multiplication, it only requires multiplication by constant $\mathsf{multConst}$, which can be performed efficiently without increasing the size of the ciphertexts and $\mathsf{SwitchKey}$. Moreover we note that line 1 of Algorithm 4 is performed before the algorithm start, i.e. we precompute coefficient vectors $\{\boldsymbol{a}_{f_d}\}_{d \in \mathbb{Z}_p}$, which appear in (3), of the interpolation polynomial.

$\mathsf{ConstEq}(\boldsymbol{C_a^{\mathsf{pow}}}, \boldsymbol{y})$. Algorithm 5 shows the sub-algorithm $\mathsf{ConstEq}(C_a^{\mathsf{pow}}, y)$. The algorithm takes a list of powers of ciphertext $\boldsymbol{C_a^{\mathsf{pow}}} := (C_a, C_a^2, C_a^3, \ldots, C_a^{p-1})$ and a plaintext input y as inputs, then homomorphically calculates a ciphertext $C_{(a=y)}$, the decryption of which gives 1 when $a = y$, or 0 when $a \neq y$. This algorithm is similar to $\mathsf{ConstDiv}$, except that the values of the precomputed coefficient vectors $\boldsymbol{a}_{f_y}$ are different. Thus, the complexity of the algorithm is almost the same as the $\mathsf{ConstDiv}$, and there is no homomorphic multiplication.

Algorithm 4 $\mathsf{ConstDiv}(\boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}}, d)$: Integer-wise Constant Division

Input: $\boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}}$: Powers of ciphertexts C_a. d: public divisor.
Output: $C_{\lfloor a/d \rfloor}$

1: (Precomputing): We define a function for dividing by the fixed constant d as

$$g_d(x) := \left\lfloor \frac{x}{d} \right\rfloor.$$

Given points $\{g_d(x)\}_{x \in \mathbb{Z}_p}$, calculate the interpolation polynomial $f_d(x)$ that satisfies $\forall x \in \mathbb{Z}_p$, $f_d(x) = g_d(x)$. For example, when plaintext modulus $p = 7$ and public divisor $d = 2$, an interpolation polynomial $f_d(x)$ is calculated given that

$$f_d(0) = 0,\ f_d(1) = 0,\ f_d(2) = 1,\ f_d(3) = 1,\ f_d(4) = 2,\ f_d(5) = 2,\ f_d(6) = 3.$$

And we obtain $f_d(x) = -2x + 3x^3 + x^5 - 2x^6$. We define $\boldsymbol{a}_{f_d}^{\mathsf{ConstDiv}} := (a_1, \ldots, a_{p-1}) = (-2, 0, 3, 0, 5, -2)$ as a coefficient vector of $f_d(x)$.

2: Output

$$C_{\lfloor a/d \rfloor} := (\boldsymbol{a}_{f_d}^{\mathsf{ConstDiv}})^\top \boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}} := \sum_{i=1}^{p-1} \mathsf{FHE.multConst}(C_a^i, a_i). \tag{3}$$

Note. We note that $\boldsymbol{a}_{f_d}^{\mathsf{ConstDiv}}$ is dependent on d, and we precompute $\boldsymbol{a}_{f_d}^{\mathsf{ConstDiv}}$ for all $d \in \mathbb{Z}_p$: We have a list of coefficient vector $\{\boldsymbol{a}_{f_0}^{\mathsf{ConstDiv}}, \ldots, \boldsymbol{a}_{f_{p-1}}^{\mathsf{ConstDiv}}\}$ as constant.

Algorithm 5 $\mathsf{ConstEq}(\boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}}, y)$: Integer-wise Constant Equality test

Input: $\boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}}$: Powers of ciphertexts C_a. y: public constant.
Output: $C_{(a=y)}$

1: (Precomputing): We define a function for testing the equality to the fixed constant y as

$$g_y(x) := \begin{cases} 1 & (x = y), \\ 0 & (x \neq y). \end{cases}$$

Given points $\{g_y(x)\}_{y \in \mathbb{Z}_p}$, calculate the interpolation polynomial $f_y(x)$ that satisfies $\forall y \in \mathbb{Z}_p$, $f_y(x) = g_y(x)$. For example, when $p = 7$ and $y = 3$, an interpolation polynomial $f_y(x)$ is calculated given that

$$f_y(0) = 0,\ f_y(1) = 0,\ f_y(2) = 0,\ f_y(3) = 1,\ f_y(4) = 0,\ f_y(5) = 0, f_y(6) = 0.$$

And we obtain $f_y(x) = 2x + 3x^2 + x^3 - 2x^4 - 3x^5 - x^6$, the coefficient vector of which is $\boldsymbol{a}_{f_y}^{\mathsf{ConstEq}} := (a_1, \ldots, a_{p-1}) = (2, 3, 1, -2, -3, -1)$.

2: Output

$$C_{(a=y)} := (\boldsymbol{a}_{f_y}^{\mathsf{ConstEq}})^\top \boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}} := \sum_{i=1}^{p-1} \mathsf{FHE.multConst}(C_a^i, a_i).$$

Note. Note that, as with $\mathsf{ConstDiv}$, we precompute the first step and we have a list $\{\boldsymbol{a}_{f_0}^{\mathsf{ConstEq}}, \ldots, \boldsymbol{a}_{f_{p-1}}^{\mathsf{ConstEq}}\}$ as constant.

Table 2: The multiplicative depth and the number of calls of Mult, Add and multConst in our algorithms for l-bit size input.

	Multiplicative Depth	Mult	Add	multConst
Pows	$O(l)$	$O(2^l)$	0	0
ConstEq	0	0	$O(2^l)$	$O(2^l)$
ConstDiv	0	0	$O(2^l)$	$O(2^l)$
Div	$O(l)$	$O(2^l)$	$O(2^{2l})$	$O(2^{2l})$

3.3 Complexity analysis and experiments

Complexity. Although Algorithm 2 might seem exhaustive and inefficient due to its `for` loop, this homomorphic computation can be performed efficiently. This is mainly because:

- ConstDiv and ConstEq do not include FHE.Mult, but include FHE.multConst. Thus, ConstDiv and ConstEq do not increase the multiplicative depth.
- The most complex part Pows is executed only once for each input ciphertext C_a and C_d before the `for` loop.

In the following, we analyse the multiplicative depth and the total complexity of our algorithm Div. Table 2 shows a summary. In our analysis we denote by l the bit length of the input integers.

Multiplicative depth. The multiplicative depth of Pows is $O(\log p) = O(l)$, as shown in Section 3.2. The multiplicative depth of Div is also $O(l)$, because we perform only one FHE.Mult after Pows in one loop in the Div algorithm, and ConstDiv and ConstEq do not include FHE.Mult. Chen *et al.* showed that the multiplicative depth of their bit-wise division algorithm [9] is $O(l^2)$. The other bit-wise division algorithms [8,28] are the same as that of [9], the idea of which is based on the non-restoring division, since their improvements are mainly for bit-wise addition and multiplication. Therefore, our algorithm provides quadratic improvement in regard to the multiplicative depth of homomorphic division.

Total complexity. It is not trivial to adequately analyse the total complexity of the FHE circuit, because the cost of the homomorphic calculation depends on the level parameter L: the order of the cost of the homomorphic calculation is $O(\lambda L^3)$, as we mentioned in Section 2.2. Moreover, as mentioned in FHE.Setup, the level L also defines the parameter m. The parameter m defines the cyclotomic polynomial $\Phi_m(x)$, which in turn defines the ciphertext space. HElib has a bound on the value L, and HElib halts if we set L too high because of the bound on the size of the cyclotomic polynomial, as noted in [9,28]. Probably based on this fact, the existing works [8, 9, 28] do not show the results on higher (> 4) bits input integers (see Table 3).

Table 3: Performance comparison.

l (bits)	p	L	nslots	Time (s)	type	method
4	2	21	720	67.94		[9]
4	2	21	720	14.63	Bit-wise	[28]
4	2	21	720	7.74		[8]
4	17	9	108	3.15		
5	37	11	340	15.11		
6	67	13	165	51.34	Integer-wise	Our (Div)
7	131	15	138	198.92		
8	257	17	396	795.84		

Although ConstDiv and ConstEq do not include homomorphic multiplication and do not increase the multiplicative depth, we perform them exhaustively in the `for` loop (from $i = 0$ to $p - 1$), and thus we cannot ignore their cost. However, since this `for` loop is parallelisable, this issue could be decreased by parallelisation. In contrast, existing secure division algorithms are not suitable for parallelisation because of their full-adder circuit (e.g. Ripple Carry adder), as discussed in [8].

We also note that the memory complexity of our algorithm is relatively high. In the precomputation of polynomial interpolation, we generate the coefficient vectors and store them as a matrix, the space for which is $\mathbb{Z}_p^{p \times p}$.

Experiments. We implemented our secure division algorithm on HElib [16], and compared timings with existing secure division algorithms based on FHE schemes. We can compare the results only for 4-bit size input, since all the existing works report only for 4-bit integer division. We also implement for higher bit values $l = 5, \ldots, 8$, and observe that the results follow our complexity analysis.

Parameters. We set the security parameter $\lambda = 80$, following the existing works of the secure division algorithm [8, 9, 28]. Let l be a the bit size of the input integer. Since our algorithm is integer-wise and stores an input integer in $\mathbb{Z}$ into $\mathbb{Z}_p$, we define the size p of the plaintext space R_p as $p = \text{nextprime}(2^l)$. We set the Hamming weight of the secret vector $w = 64$. For the level parameter L, which is related to the multiplicative depth of the circuit, we search the minimum level by trial and error. As L is the lower level, we can perform the circuit faster. However, setting L too small leads to incorrectness of the outputs. For the rest of the parameter including "nslots", which means the number of the SIMD slots, we use the default values automatically calculated by HElib. For further details, we refer the reader to [16].

Results. Table 3 shows our timing results, in addition to results given in existing work for bit-wise integer division. All timings were generated on a PC with a

Algorithm 6 $\mathsf{ConstFunc}(\boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}}, y, g(\cdot,\cdot))$: Integer-wise Constant Function

Input: $\boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}}$: Powers of ciphertexts C_a. y: public constant. $g(\cdot,\cdot)$: 2-variable function.
Output: $C_{g(a,y)} = g(C_a, y)$

1: (Precomputing): In this algorithm, fix y. Thus, we consider the input 2-variable function $g(x,y) : \mathbb{Z}_p \times \mathbb{Z}_p \to \mathbb{Z}_p$ as 1-variable function $g_y(x) := g(x,y)$. For a fixed y, an interpolation polynomial $f_y(x)$ for $g_y(x)$ is calculated given the values $g(0,y), g(1,y), \ldots, g(p-1,y)$. And we can write the interpolation polynomial as follows: $f_y(x) = a_{\{1,y\}} \cdot x + a_{\{2,y\}} \cdot x^2 + a_{\{3,y\}} \cdot x^3 + \cdots + a_{\{p-1,y\}} \cdot x^6 \bmod p$.
2: Calculate $C_{g(a,y)} = f_y(C_a) = \boldsymbol{a}_{f_y}^{\top} \boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}} \bmod p$, where $\boldsymbol{a}_{f_y} = \{a_{\{1,y\}}, a_{\{2,y\}}, a_{\{3,y\}}, \ldots, a_{\{p-1,y\}}\}$.

3.4 GHz Intel Core i5 and 16 GB RAM. To the best of our knowledge, the work by Chen *et al.* [8] provides the fastest results for 4-bit size input integers; thus our algorithm is the fastest secure division algorithm. While existing works report only for $l = 4$, we implemented our algorithm also for the higher bit sizes $l = 5, \ldots, 8$. We can observe that our algorithm requires only $L = O(l)$, following our analysis given in Section 3.3. Moreover, we can observe that required L of our algorithm is lower than that of the bit-wise algorithm for $l = 4$. Based on this fact, and that the bit-wise algorithm requires $L = O(l^2)$, we can expect that L of our algorithm is globally less than that of the bit-wise algorithm.

We also note that nslots of our algorithm, which are automatically calculated by HElib depending on the other parameters, is lower than in bit-wise algorithms. This means that the amortised cost of our algorithm might be larger than existing algorithms.

4 Integer-wise Homomorphic Evaluation of Arbitrary 2-variable Function

We show that our secure division algorithm can be generalised to integer-wise secure computation of any 2-variable function with the same computation cost.

4.1 Algorithms

Our integer-wise homomorphic evaluation algorithm of any (predefined) 2-variable function $\mathsf{Func}(C_a, C_d)$ given in Algorithm 7 is constructed based on Div, by replacing $\mathsf{ConstDiv}$ with $\mathsf{ConstFunc}(\boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}}, y, g(\cdot,\cdot))$ (Algorithm 6), which performs 1-variable function $g_y(x) := g(x,y)$ over a ciphertext C_a.

ConstFunc. Algorithm 6 shows the $\mathsf{ConstFunc}$ algorithm. The algorithm takes a list of powers of ciphertext $\boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}} := (C_a, C_a^2, C_a^3, \ldots, C_a^{p-1})$, a plaintext y and a 2-variable function $g(\cdot,\cdot)$ (i.e., coefficient vectors of its interpolation polynomial), and then homomorphically calculates a ciphertext $C_{g(a,y)}$, the decryption of which gives $g(a,y)$. Recall that, in the $\mathsf{ConstDiv}(C_a^{\mathsf{pow}}, d)$, we used polynomial

Algorithm 7 $\mathsf{Func}(C_a, C_b, g(\cdot,\cdot))$: Integer-wise Homomorphic 2-variable function

Input: Ciphertexts C_a, C_b. 2-variable function g.
Output: $C_{g(a,b)} = g(C_a, C_b)$

1: $C_{\mathsf{sum}} = 0$
2: $\boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}} = \mathsf{Pows}(C_a)$ ▷ $\boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}} := \{C_a, C_a^2, C_a^3, \ldots, C_a^{p-1}\}$
3: $\boldsymbol{C}_{\boldsymbol{b}}^{\mathsf{pow}} = \mathsf{Pows}(C_b)$ ▷ $\boldsymbol{C}_{\boldsymbol{b}}^{\mathsf{pow}} := \{C_b, C_b^2, C_b^3, \ldots, C_b^{p-1}\}$
4: **for** $i = 1$ to $(p-1)$ **do**
5: $\quad C_{g(a,i)} = \mathsf{ConstFunc}(\boldsymbol{C}_{\boldsymbol{a}}^{\mathsf{pow}}, i, g)$ ▷ $C_{g(a,i)} = g(C_a, i)$
6: $\quad C_{(i=b)} = \mathsf{ConstEq}(\boldsymbol{C}_{\boldsymbol{b}}^{\mathsf{pow}}, i)$ ▷ $C_{(i=b)} = C_1$ if $i = b$, $C_{(i=b)} = C_0$ otherwise.
7: $\quad C_{\mathsf{sum}} = C_{\mathsf{sum}} + \mathsf{FHE.Mult}(C_{g(a,i)}, C_{(i=b)})$ ▷ $\mathsf{FHE.Mult}(C_{g(a,i)}, C_{(i=b)}) = C_{g(a,b)}$ if $i = b$, C_0 otherwise.
8: **end for**
9: Output $C_{g(a,b)} = C_{\mathsf{sum}}$

interpolation to construct a function $f_d(x) = \lfloor \frac{x}{d} \rfloor$ for all $d \in \mathbb{Z}_p$, giving data points $\{f_d(0), f_d(1), \ldots, f_d(p-1)\}$. We can simply generalise ConstDiv as a general function: in ConstFunc, we use polynomial interpolation to construct a required function $g_y(x)$ defined by given data points $\{g_y(0), g_y(1), \ldots, g_y(p-1)\}$, for all $y \in \mathbb{Z}_p$. Only the data points (or, coefficient vectors) are different between ConstDiv and ConstFunc. Thus, the order of multiplicative depth of ConstFunc is exactly the same as ConstDiv. Note that we precompute the coefficient vectors $\boldsymbol{a}_{f_y}$ for all $y \in \mathbb{Z}_p$ using the polynomial interpolation, as with ConstDiv.

Func. Algorithm 7 shows our algorithm for integer-wise homomorphic evaluation of the arbitrary 2-variable function, Func. The algorithm takes two ciphertexts C_a and C_b, which are ciphertexts of a and b, respectively, and a 2-variable function $g(\cdot,\cdot)$ (i.e., coefficient vectors of its interpolation polynomial) as inputs. It then homomorphically calculates a ciphertext $C_{g(a,b)}$, which decrypts to $g(a,b)$. This algorithm is almost the same as Div (Algorithm 2), except that ConstDiv is replaced by ConstFunc. Since the order of multiplicative depth of ConstFunc is exactly the same as ConstDiv, that of Func is exactly the same as Div.

5 Conclusion

We propose a first secure integer-wise homomorphic division algorithm on fully homomorphic encryption schemes. We implemented the algorithm on HElib, and show that our algorithm is over twice as fast as the fastest existing algorithm given in [8]. We also showed that the multiplicative depth of our algorithm is $O(l)$ for l-bit size integer, while that of existing division algorithms is $O(l^2)$.

Furthermore, we generalise our secure division algorithm and propose an algorithm for secure calculation of general 2-variable functions. We showed that the complexity of the algorithm is the same as our division algorithm. This means that the homomorphic calculation of any 2-variable functions taking integer inputs can be performed with multiplicative depth $O(l)$.

References

1. Bost, R., Popa, R.A., Tu, S., Goldwasser, S.: Machine Learning Classification over Encrypted Data. In: NDSS Symposium 2015. p. 04_1_2. Internet Society (2015). https://doi.org/10.14722/ndss.2015.23241
2. Bourse, F., Minelli, M., Minihold, M., Paillier, P.: Fast Homomorphic Evaluation of Deep Discretized Neural Networks. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018. pp. 483–512. Springer (2018). https://doi.org/10.1007/978-3-319-96878-0_17
3. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) Fully Homomorphic Encryption Without Bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. pp. 309–325. ITCS '12, ACM (2012). https://doi.org/10.1145/2090236.2090262
4. Brakerski, Z., Vaikuntanathan, V.: Efficient Fully Homomorphic Encryption from (Standard) LWE. In: Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science. pp. 97–106. FOCS '11, IEEE Computer Society (2011). https://doi.org/10.1109/FOCS.2011.12
5. Brakerski, Z., Vaikuntanathan, V.: Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. In: Rogaway, P. (ed.) Advances in Cryptology – CRYPTO 2011. pp. 505–524. Springer (2011). https://doi.org/10.1007/978-3-642-22792-9_29
6. Brakerski, Z., Vaikuntanathan, V.: Lattice-based FHE As Secure As PKE. In: Proceedings of the 5th Conference on Innovations in Theoretical Computer Science. pp. 1–12. ITCS '14, ACM (2014). https://doi.org/10.1145/2554797.2554799
7. Çetin, G.S., Doröz, Y., Sunar, B., Savaş, E.: Depth Optimized Efficient Homomorphic Sorting. In: Lauter, K., Rodríguez-Henríquez, F. (eds.) Progress in Cryptology – LATINCRYPT 2015. pp. 61–80. Springer (2015). https://doi.org/10.1007/978-3-319-22174-8_4
8. Chen, J., Feng, Y., Liu, Y., Wu, W.: Faster Binary Arithmetic Operations on Encrypted Integers. In: the 7th International Workshop on Computer Science and Engineering (2017)
9. Chen, Y., Gong, G.: Integer arithmetic over ciphertext and homomorphic data aggregation. In: 2015 IEEE Conference on Communications and Network Security (CNS). pp. 628–632 (Sept 2015). https://doi.org/10.1109/CNS.2015.7346877
10. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully Homomorphic Encryption over the Integers. In: Gilbert, H. (ed.) Advances in Cryptology – EUROCRYPT 2010. pp. 24–43. Springer (2010). https://doi.org/10.1007/978-3-642-13190-5_2
11. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University (2009). crypto.stanford.edu/craig
12. Gentry, C.: Fully Homomorphic Encryption Using Ideal Lattices. In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing. pp. 169–178. STOC '09, ACM (2009). https://doi.org/10.1145/1536414.1536440
13. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic Evaluation of the AES Circuit. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology – CRYPTO 2012. pp. 850–867. Springer (2012). https://doi.org/10.1007/978-3-642-32009-5_49
14. Gentry, C., Sahai, A., Waters, B.: Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology – CRYPTO 2013. pp. 75–92. Springer (2013). https://doi.org/10.1007/978-3-642-40041-4_5

15. Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In: Balcan, M.F., Weinberger, K.Q. (eds.) Proceedings of The 33rd International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 48, pp. 201–210. PMLR (20–22 Jun 2016). http://proceedings.mlr.press/v48/gilad-bachrach16.html
16. Halevi, S., Shoup, V.: HElib - An Implementation of homomorphic encryption. https://github.com/shaih/HElib/
17. Halevi, S., Shoup, V.: Algorithms in HElib. In: Garay, J.A., Gennaro, R. (eds.) Advances in Cryptology – CRYPTO 2014. pp. 554–571. Springer (2014). https://doi.org/10.1007/978-3-662-44371-2_31
18. Halevi, S., Shoup, V.: Bootstrapping for HElib. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology – EUROCRYPT 2015. pp. 641–670. Springer (2015). https://doi.org/10.1007/978-3-662-46800-5_25
19. Halevi, S., Shoup, V.: Faster Homomorphic Linear Transformations in HElib. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018. pp. 93–120. Springer (2018). https://doi.org/10.1007/978-3-319-96884-1_4
20. Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 1651–1669. USENIX Association, Baltimore, MD (2018). https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar
21. Lyubashevsky, V., Peikert, C., Regev, O.: On Ideal Lattices and Learning with Errors over Rings. In: Gilbert, H. (ed.) Advances in Cryptology – EUROCRYPT 2010. pp. 1–23. Springer (2010). https://doi.org/10.1007/978-3-642-13190-5_1
22. Narumanchi, H., Goyal, D., Emmadi, N., Gauravaram, P.: Performance Analysis of Sorting of FHE Data: Integer-Wise Comparison vs Bit-Wise Comparison. In: 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA). pp. 902–908 (March 2017). https://doi.org/10.1109/AINA.2017.85
23. Regev, O.: On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. J. ACM **56**(6), 34:1–34:40 (Sep 2009). https://doi.org/10.1145/1568318.1568324
24. Shoup, V.: NTL: A Library for doing Number Theory. http://shoup.net/ntl/
25. Smart, N.P., Vercauteren, F.: Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes. In: Nguyen, P.Q., Pointcheval, D. (eds.) Public Key Cryptography – PKC 2010. pp. 420–443. Springer (2010). https://doi.org/10.1007/978-3-642-13013-7_25
26. Smart, N.P., Vercauteren, F.: Fully homomorphic SIMD operations. Designs, Codes and Cryptography **71**(1), 57–81 (Apr 2014). https://doi.org/10.1007/s10623-012-9720-4
27. Stehlé, D., Steinfeld, R.: Faster Fully Homomorphic Encryption. In: Abe, M. (ed.) Advances in Cryptology - ASIACRYPT 2010. pp. 377–394. Springer (2010). https://doi.org/10.1007/978-3-642-17373-8_22
28. Xu, C., Chen, J., Wu, W., Feng, Y.: Homomorphically Encrypted Arithmetic Operations Over the Integer Ring. In: Bao, F., Chen, L., Deng, R.H., Wang, G. (eds.) Information Security Practice and Experience. pp. 167–181. Springer (2016). https://doi.org/10.1007/978-3-319-49151-6_12