Efficient Homomorphic Evaluation of Arbitrary Uni/Bivariate Integer Functions and Their Applications*

Daisuke Maeda[1], Koki Morimura[1], Shintaro Narisada[2], Kazuhide Fukushima[2], and Takashi Nishide[1]

[1] University of Tsukuba, Japan
nishide@risk.tsukuba.ac.jp
[2] KDDI Research, Inc., Saitama, Japan.

Abstract. We propose how to homomorphically evaluate arbitrary univariate and bivariate integer functions such as division. A prior work proposed by Okada et al. (WISTP'18) uses polynomial evaluations such that the scheme is still compatible with the SIMD operations in BFV and BGV, and is implemented with the input domain size $\mathbb{Z}_{257}$. However, the scheme of Okada et al. requires the quadratic number of plaintext-ciphertext multiplications and ciphertext-ciphertext additions in the input domain size, and although these operations are more lightweight than the ciphertext-ciphertext multiplication, the quadratic complexity makes handling larger inputs quite inefficient.

In this work, first we improve the prior work and also propose a new approach that exploits the packing method to handle the larger input domain size instead of enabling the SIMD operation, thus making it possible to work with the larger input domain size, e.g., $\mathbb{Z}_{2^{15}}$ in a reasonably efficient way. In addition, we show how to slightly extend the input domain size to $\mathbb{Z}_{2^{16}}$ with a relatively moderate overhead. Further we show another approach to handling the larger input domain size by using two ciphertexts to encrypt one integer plaintext and applying our techniques for uni/bivariate function evaluation.

We implement the prior work of Okada et al., our improved scheme of Okada et al., and our new scheme in PALISADE with the input domain size $\mathbb{Z}_{2^{15}}$, and confirm that the estimated run-times of the prior work and our improved scheme of the prior work are still about 117 days and 59 days respectively while our new scheme can be computed in 307 seconds.

Keywords: Fully Homomorphic Encryption, Polynomial Interpolation, Homomorphic Evaluation of Non-Linear Bivariate Function

* A preliminary version of this work appeared in WAHC'22 [MMN22, https://doi.org/10.1145/3560827.3563378]. In this extended version, we add to Appendix A how to extend the input domain size from $\mathbb{Z}_t$ to $\mathbb{Z}_{t^2}$ by using two ciphertexts to encrypt one integer plaintext and how to realize basic integer operations with the extended input domain $\mathbb{Z}_{t^2}$.

1 Introduction

1.1 Background

Fully homomorphic encryption (FHE) is a promising tool for achieving privacy in the data analysis, and has the advantage that it enables non-interactive secure computation compared with, e.g., secret sharing based secure computation. After the first FHE construction of Gentry [Gen09], many FHE schemes are proposed and already implemented in modern software libraries like [HS14,PAL20,SEA22]. FHE schemes can be categorized into three classes. The first class deals with Boolean circuits and lookup tables based on functional bootstrapping and the FHEW and TFHE (also known as CGGI) schemes [DM15,CGGI20,GBA21,CLOT21] are included in this class. The second class can encrypt vectors, and supports modular arithmetic over a finite field in each slot of the vectors (called SIMD functionality), which is typically used to simulate integer arithmetic. The Brakerski-Gentry-Vaikuntantan (BGV) and Brakerski-Fan-Vercauteren (BFV) schemes [BGV12,Bra12,FV12,KPZ21] are included in this class. The third class supports approximate computation of vectors consisting of real and complex numbers, and the Cheon-Kim-Kim-Song (CKKS) scheme [CKKS17] is included in this class. The security of these classes is based on Ring Learning With Errors (RLWE), and each of these FHE schemes can be useful depending on the types of computation we need to perform securely. In general, the FHE schemes in the first class are the most versatile, and for example, in [GBA21], any computation can be performed by representing it as a lookup table $(\mathbb{Z}_B)^n \to \mathbb{Z}_B$ where B is a small radix (e.g., $\leq 2^6$) and by applying the technique called "functional bootstrap" iteratively to select the final output from the lookup table[3]. On the other hand, the second and third classes can be more suitable for integer/fixed-point arithmetic computation including many addition and multiplication operations.

In this work, we focus on the BFV scheme of the second class[4]. In the BFV scheme supporting integer-wise operations, addition and multiplication can be performed easily without the bit-wise/digit-wise encryption approach, and if we can realize the mixed computation in which these addition and multiplication can be combined with complex non-linear functions seamlessly, it will be advantageous. One standard way to compute a non-linear function f in the second and third classes is to perform polynomial evaluation by representing f as a polynomial via polynomial interpolation. If the non-linear function f we want to homomorphically evaluate is univariate and the input domain size is small, a simple approach based on polynomial evaluation is viable, but if f is bivariate and the input domain size N is relatively large (e.g., $N = 2^{15}$), it can be non-trivial to compute an arbitrary non-linear function $f\colon \mathbb{Z}_N \times \mathbb{Z}_N \to \mathbb{Z}_N$ in a reasonably efficient way even if we use polynomial evaluation.

[3] As mentioned in [GBA21], evaluating lookup tables homomorphically inherently requires the exponential time complexity in the input size.

[4] We believe most of our idea can be used for BGV as well.

1.2 Our Contributions

First we improve the prior work of Okada et al. [OCHK18] that shows how to homomorphically compute an arbitrary bivariate integer function based on polynomial evaluation. Our improved scheme of [OCHK18] has the advantage that it can still be combined with the SIMD functionality [SV14] of BFV/BGV, i.e., the parallel computation of the same bivariate function in multiple slots is supported, but as our experiment shows, it is still prohibitively inefficient with, e.g., our typical input domain size [5] $N = 2^{15}$ and plaintext modulus $t = 2^{16} + 1$.

To overcome the above issue, we also propose another approach at the price of allowing only a single slot to be used during the computation of bivariate functions. That is, the SIMD functionality of BFV/BGV is usually used for parallel computation, but instead we exploit the SIMD functionality to realize an arbitrary non-linear univariate function $f\colon \mathbb{Z}_N \to \mathbb{Z}_N$, which we call One-HotSlot technique. Further combining One-HotSlot with the Paterson-Stockmeyer method [PS73], we show how to compute an arbitrary bivariate non-linear function $f\colon \mathbb{Z}_N \times \mathbb{Z}_N \to \mathbb{Z}_N$, which is much faster than homomorphically computing the bivariate function in all the slots in parallel (i.e., with fully packed ciphertexts)[6]. Further we show that the input domain size can be extended such that the bivariate function is of type $\mathbb{Z}_{2N} \times \mathbb{Z}_{2N} \to \mathbb{Z}_{2N}$ with a relatively moderate overhead. In Appendix A, we slightly extend our above techniques for uni/bivariate function evaluation so that they can handle functions of type $\mathbb{Z}_t \to \mathbb{Z}_t$ and $\mathbb{Z}_t \times \mathbb{Z}_t \to \mathbb{Z}_t$ where t is a plaintext modulus, and show another approach to extending the input domain size from $\mathbb{Z}_t$ to $\mathbb{Z}_{t^2}$ by using two ciphertexts to encrypt one integer plaintext.

1.3 Related Work

Computing non-linear arbitrary functions including integer-wise comparison and division can be a challenging task in the context of CKKS, BFV, and BGV, and polynomial interpolation and evaluation are the important tools for realizing non-linear functions over a prime finite field or ring and even for performing bootstrap procedures [CH18,HS21].

Lu et al. [LZS18] proposed a homomorphic comparison operation for BFV/BGV, and it was extended by Ishimaki and Yamana [IY18]. The underlying idea is to encode a plaintext integer a as a polynomial X^a before encrypting it by FHE. Although such an encoding can achieve better efficiency for a comparison operation, the special-purpose encoding makes it inefficient to perform ciphertext-ciphertext additions and multiplications.

Kaji et al. [KMNN19] investigated how to represent non-linear max and argmax functions as concrete bivariate polynomials over a prime finite field $\mathbb{F}_p$ in relation to a homomorphic comparison operation, but their results are not

[5] As shown in §4.2, our estimate shows that it requires about 59 days in our experimental environment.

[6] It takes 307 seconds (§4.2) compared with 59 days of our improved scheme of [OCHK18] in the setting of our typical input domain size 2^{15}.

so efficient in the sense that $O(p^2)$ ciphertext-ciphertext multiplications are required.

Tan et al. [TLW+20] proposed the special-purpose comparison operation using BGV and equality function proposed by Kim et al. [KLLW16]. Roughly speaking, the input integers are represented as vectors in the base-p representation, and the vectors are encrypted by using the SIMD functionality and compared in lexicographical order by using the bivariate polynomial.

Cheon et al. [CKK20] proposed the comparison function for CKKS. Roughly speaking, based on composite polynomial approximation, they represent the comparison function f by finding f' such that $f'^{(d)} = f' \circ f' \circ \cdots \circ f'$ gets closer to f by increasing d. Here the computation is approximate and the inputs to the comparison function need to be within a specific range, and allowing the two inputs to be close to each other can increase the computational cost of the comparison function.

Iliashenko and Zucca [IZ21] proposed how to represent the comparison function as concrete bivariate and univariate polynomials over a prime finite field $\mathbb{F}_p$ where BFV and BGV are the underlying FHE schemes. Also they showed that polynomial evaluations can be done with $O(p)$ ciphertext-ciphertext multiplications in the case of the bivariate polynomial, and with $O(\sqrt{p})$ ciphertext-ciphertext multiplications in the case of the univariate polynomial. For homomorphic comparison of large inputs, in [IZ21], input integers need to be represented in the base-p representation, encoded as elements in the extension field $\mathbb{F}_{p^d}$ assuming p is not large, and the corresponding vectors of of coefficients are compared in lexicographical order, so the comparison operation does not seem to be combined with homomorphic additions and multiplications seamlessly to simulate integer arithmetic.

Iliashenko et al. [INZ21] showed that several non-linear univariate functions such as "modulo", "is power of b", "Hamming weight" and "Mod2" can have nice polynomial structures like an odd function when the related parameters satisfy specific conditions[7], which allows more efficient polynomial evaluation compared with the well-known Paterson-Stockmeyer method [PS73].

Okada et al. [OCHK18] proposed how to compute the division function using BGV by combining the polynomial evaluations of two univariate functions, equality function and division function with a public divisor, and showed that 8-bit integer division can be computed in 795.8 seconds. Their method to compute the division function can easily be generalized to realize arbitrary bivariate integer functions.

To realize arbitrary bivariate integer functions rather than specific or special-purpose non-linear functions, we improve the work of [OCHK18] and also propose a new approach enabling to handle the larger input domain size.

[7] For example, a "modulo m" function can be represented by a univariate polynomial similar to an odd polynomial when $p \equiv m - 1 \bmod m$.

2 Preliminaries

2.1 Notation

We summarize the symbols used in this work in Table 1. One of the typical settings for this work is $N = 2^{15}$ and $t = 2N + 1 = 65537$ [8].

Table 1. Notation

Notation	Description
N	Power of two
$t,\ q$	Integers for plaintext and ciphertext moduli where $t \ll q$ and t is a prime s.t. $2N \mid t-1$
$\mathbb{Z}[X]$	Set of integer coefficient polynomials of variable X
$\mathcal{R}$	Ring $\mathbb{Z}[X]/(X^N+1)$
$\mathcal{R}_t$	Ring defining plaintext space $\mathbb{Z}_t[X]/(X^N+1)$
$\mathcal{R}_q$	Ring defining ciphertext space $\mathbb{Z}_q[X]/(X^N+1)$
$[\![M]\!]$	Ciphertext of plaintext M
$\oplus,\ \ominus,\ \otimes$	Homomorphic addition, subtraction and multiplication
$L[j]$	j-th element of list/array L with zero-based index

2.2 Packing Method in Fully Homomorphic Encryption

The BGV [BGV12] and BFV [FV12] schemes support integers operations and CKKS [CKKS17] scheme supports fixed-point arithmetic. In this work, we focus on the BFV scheme [FV12], which is one of the schemes that support the following functionalities: integer-wise addition/multiplication, Galois automorphism, and packing method.

2.2.1 Packing Method

The FHE schemes based on Ring-LWE can pack a set of integers into a single plaintext or ciphertext polynomial by setting some variables appropriately [SV14]. Let t and N be such that $2N \mid t-1$, and g be a generator of $\mathbb{Z}_t$ satisfying $g^{t-1} \equiv 1 \bmod t$. Then $\omega \equiv g^{\frac{t-1}{2N}} \in \mathbb{Z}_t$ is the primitive $2N$-th root of 1 in $\mathbb{Z}_t$, i.e., $\omega^{2N} \equiv 1 \mod t$.

In this setting, a ciphertext of a polynomial $f(X) = a_0 + a_1 X + \cdots + a_{N-1}X^{N-1} \in \mathcal{R}_t$ can be viewed as a ciphertext that packs a plaintext vector $(a'_0,\ a'_1, \ldots, a'_{N-1}) \in \mathbb{Z}_t^N$ determined by the following matrix W.

[8] Another possible example is $N = 2^{15}$ and $t = 3 \times 2^{18} + 1$. In this setting, the ciphertext size becomes larger and it causes lower performance, but meanwhile we can have a larger level parameter (multiplicative depth).

$$\underbrace{\begin{bmatrix} a'_0 \\ a'_1 \\ a'_2 \\ \vdots \\ a'_{N-1} \end{bmatrix}}_{\mathbb{Z}_t^N} = \underbrace{\begin{bmatrix} 1 & \omega & \cdots & \omega^{N-1} \\ 1 & \omega^3 & \cdots & \left(\omega^3\right)^{N-1} \\ 1 & \omega^5 & \cdots & \left(\omega^5\right)^{N-1} \\ & & \vdots & \\ 1 & \omega^{2N-1} & \cdots & \left(\omega^{2N-1}\right)^{N-1} \end{bmatrix}}_{W \in \mathbb{Z}_t^{N \times N}} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{N-1} \end{bmatrix}}_{\mathbb{Z}_t^N}$$

$$= \begin{bmatrix} f(\omega) \\ f\left(\omega^3\right) \\ f\left(\omega^5\right) \\ \vdots \\ f\left(\omega^{2N-1}\right) \end{bmatrix} = \begin{bmatrix} a_0 + a_1\omega + \cdots + a_{N-1}\omega^{N-1} \\ a_0 + a_1\omega^3 + \cdots + a_{N-1}\left(\omega^3\right)^{N-1} \\ a_0 + a_1\omega^5 + \cdots + a_{N-1}\left(\omega^5\right)^{N-1} \\ \vdots \\ a_0 + a_1\omega^{2N-1} + \cdots + a_{N-1}\left(\omega^{2N-1}\right)^{N-1} \end{bmatrix}$$

Here the packed ciphertext is denoted by $[\![f(X)]\!] = [\![(a'_0,\ a'_1, \ldots, a'_{N-1})]\!]$, and similarly, the packed plaintext is denoted by $[f(X)] = [(a'_0,\ a'_1, \ldots, a'_{N-1})]$. We call the i-th element of a vector the i-th slot. Here the addition and multiplication of vectors are performed element-wise on each slot. I.e., when we consider two packed ciphertexts $[\![f(X)]\!] = [\![(a'_0, a'_1, \ldots, a'_{N-1})]\!]$ and $[\![g(X)]\!] = [\![(b'_0, b'_1, \ldots, b'_{N-1})]\!]$, it holds that

$$[\![f(X)]\!] \oplus [\![g(X)]\!] = [\![(a'_0 + b'_0, a'_1 + b'_1, \ldots, a'_{N-1} + b'_{N-1})]\!]$$
$$[\![f(X)]\!] \otimes [\![g(X)]\!] = [\![(a'_0 b'_0, a'_1 b'_1, \ldots, a'_{N-1} b'_{N-1})]\!]$$

where the computation in each slot is done modulo t.

2.3 Polynomial Interpolation

Polynomial interpolation is a method to derive a polynomial f satisfying $y_i = f(x_i)$ for $0 \leq i \leq N-1$, given N points $\{(x_i, y_i)\}$ where $x_i \neq x_j$ if $i \neq j$, and $f(x)$ is obtained by the following equation [9].

$$f(x) \equiv \sum_{i=0}^{N-1} \left(\prod_{0 \leq j \leq N-1, j \neq i} \frac{x - x_j}{x_i - x_j} \right) \cdot y_i \bmod t$$

2.4 Paterson-Stockmeyer Method

Given polynomial $f(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{N-1} x^{N-1}$, the naive method to evaluate $f(x)$ requires computing powers $x, x^2, \cdots, x^{N-1}$, leading to $O(N)$ multiplications. However, by using the well-known Paterson-Stockmeyer method

[9] We note that if t is not a prime, polynomial interpolation may not work.

(PS method) [PS73], the number of multiplications required for polynomial evaluation can decrease to $O(\sqrt{N})$. In the PS method, $f(x)$ is transformed as follows, and evaluated with precomputed $\{x, x^2, \ldots, x^{p-1}\}$ and $\{x^p, (x^p)^2, \ldots, (x^p)^{s-1}\}$ where $p, s \approx \sqrt{N}$.

$$\begin{aligned} f(x) =& c_0 + c_1 x + \cdots + c_{N-1} x^{N-1} \quad (\text{where } N = ps) \\ =& \left(c_0 + c_1 x + \cdots + c_{p-1} x^{p-1}\right) \\ &+ \left(c_p + c_{p+1} x + \cdots + c_{2p-1} x^{p-1}\right) \times x^p \\ &+ \left(c_{2p} + c_{2p+1} x + \cdots + c_{3p-1} x^{p-1}\right) \times (x^p)^2 \\ &\vdots \\ &+ \left(c_{(s-1)p} + c_{(s-1)p+1} x + \cdots + c_{(s-1)p+p-1} x^{p-1}\right) \times (x^p)^{s-1} \end{aligned}$$

3 Proposed Schemes

First we recall the prior work of Okada et al. [OCHK18] that shows how to compute an arbitrary bivariate integer function with BFV/BGV by using polynomial evaluations. Next we show how to improve [OCHK18] by reducing the number of polynomial evaluations. While the prior work [OCHK18] and our improved variant of [OCHK18] can work with the SIMD functionality to enable parallel computation, as shown in our experiment (§4.2), these are quite inefficient when the input domain size becomes relatively large. To address this issue, we also propose another approach that exploits the packing method to handle the larger input domain size instead of enabling the SIMD operation.

3.1 Prior Method for Homomorphically Computing Bivariate Function Based on Polynomial Evaluation

Here we recall the homomorphic integer-wise division [OCHK18] (Algorithm 2). As in [OCHK18], for simplicity, we describe Algorithm 2 assuming that the plaintext is a scalar rather than a vector for the SIMD operation, but Algorithm 2 can easily be adapted to work with the SIMD functionality. We consider computing the division $\lfloor \frac{a}{d} \rfloor$ where $a, d \in \mathbb{Z}_t$[10]. First we can precompute the coefficients of polynomials $f_i(x)$[11], $g_i(x) \colon \mathbb{Z}_t \to \mathbb{Z}_t$ satisfying

$$f_i(x) = \left\lfloor \frac{x}{i} \right\rfloor, \quad g_i(x) = \begin{cases} 1 & (\text{if } x = i) \\ 0 & (\text{otherwise}) \end{cases}$$

via polynomial interpolation. Next we compute the ciphertexts of powers $C_a^{pow} \leftarrow \mathsf{Pows}(\llbracket a \rrbracket, t)$, $C_d^{pow} \leftarrow \mathsf{Pows}(\llbracket d \rrbracket, t)$ necessary for polynomial evaluation with

[10] Here we use the division operation as an example of a bivariate function, but actually any bivariate function $\mathbb{Z}_t \times \mathbb{Z}_t \to \mathbb{Z}_t$ can be computed.

[11] Here we use the division as an example, but by changing $f_i(x)$, an arbitrary bivariate function can actually be computed.

multiplicative depth $O(\log_2(t))$ where Pows is given in Algorithm 1. In ConstDiv and ConstEq, we obtain $[\![\lfloor \frac{a}{i} \rfloor]\!]$ and $[\![i = d]\!]$ respectively by homomorphically computing the inner products between the powers and the coefficients of $f_i(x)$, $g_i(x)$[12]. We note that $[\![i = d]\!]$ equals $[\![1]\!]$ if $i = d$, and $[\![0]\!]$ otherwise, which means that $[\![\lfloor \frac{a}{i} \rfloor]\!] \otimes [\![i = d]\!]$ equals $[\![\lfloor \frac{a}{d} \rfloor]\!]$ when $i = d$ and $[\![0]\!]$ otherwise. Finally we obtain the encrypted division result $[\![\lfloor \frac{a}{d} \rfloor]\!]$ in S.

Algorithm 1 Pows

Input: $[\![a]\!]$, u
1: $k = \lfloor \log(u) \rfloor$
2: **for** $i = 0$ to $(k-1)$ **do**
3: **for** $j = 1$ to 2^i **do**
4: $[\![a^{2^i+j}]\!] \leftarrow [\![a^{2^i}]\!] \otimes [\![a^j]\!]$
5: **end for**
6: **end for**
7: **if** $2^k < u - 1$ **then**
8: **for** $i = 1$ to $u - 1 - 2^k$ **do**
9: $[\![a^{2^k+i}]\!] \leftarrow [\![a^{2^k}]\!] \otimes [\![a^i]\!]$
10: **end for**
11: **end if**
Output: $C_a^{pow} = ([\![a^0]\!], [\![a]\!], [\![a^2]\!], \ldots, [\![a^{u-1}]\!])$

Algorithm 2 Homomorphic Division $[\![\lfloor \frac{a}{d} \rfloor]\!]$

Input: $[\![a]\!]$, $[\![d]\!]$
1: $S \leftarrow [\![0]\!]$
2: $C_a^{pow} \leftarrow \mathsf{Pows}([\![a]\!], t)$, $C_d^{pow} \leftarrow \mathsf{Pows}([\![d]\!], t)$
3: **for** $i = 0$ to $t - 1$ **do**
4: $[\![\lfloor \frac{a}{i} \rfloor]\!] = \mathsf{ConstDiv}(C_a^{pow}, i)$
5: $[\![i = d]\!] = \mathsf{ConstEq}(C_d^{pow}, i)$
6: $S \leftarrow S \oplus [\![\lfloor \frac{a}{i} \rfloor]\!] \otimes [\![i = d]\!]$
7: **end for**
Output: $S = [\![\lfloor \frac{a}{d} \rfloor]\!]$

3.2 Reducing the Number of Polynomial Evaluations

In [OCHK18], polynomial evaluations (corresponding to computing the inner products between the powers and polynomial coefficients) are performed in both ConstDiv for division and ConstEq for equality check. Although the polynomial

[12] For this polynomial evaluation, computing all the necessary powers first is more efficient than using the PS method. It is because the polynomial evaluation is iterated t times and in each iteration, we can go without ciphertext-ciphertext multiplications.

evaluations do not include ciphertext-ciphertext multiplications, this computation is the dominant part of the whole computation due to t iterations in the for loop. Algorithm 2 of [OCHK18] is natural and seems optimal, but we show that the number of polynomial evaluations can be reduced further by half.

3.2.1 Precomputation

We consider the division $\left\lfloor \frac{a}{d} \right\rfloor$ where $a, d \in \mathbb{Z}_{t'}$ and $t' \leq t$ (i.e., t' is the input domain size and t is the plaintext modulus). If d is fixed, we can precompute the coefficients $(c_{0,d},\ c_{1,d},\ c_{2,d},\ \ldots,\ c_{t'-1,d})$ of polynomial $f_d(x) \colon \mathbb{Z}_{t'} \to \mathbb{Z}_{t'}$

$$f_d(x) = \left\lfloor \frac{x}{d} \right\rfloor = c_{0,d} + c_{1,d}x + c_{2,d}x^2 + \cdots + c_{t'-1,d}x^{t'-1} \bmod t$$

via polynomial interpolation. Also for $0 \leq j \leq t' - 1$ we define polynomials $g_j(x) \colon \mathbb{Z}_{t'} \to \mathbb{Z}_{t'}$

$$g_j(x) = c'_{0,j} + c'_{1,j}x + c'_{2,j}x^2 + \cdots + c'_{t'-1,j}x^{t'-1} \bmod t$$

that, given d as x, returns the j-th coefficient $c_{j,d}$ of $f_d(x)$. Now let V, Y, D and Z be

$$V = \begin{bmatrix} 0^0 & 0^1 & 0^2 & \cdots & 0^{t'-1} \\ 1^0 & 1^1 & 1^2 & \cdots & 1^{t'-1} \\ & & & \vdots & \\ (t'-1)^0 & (t'-1)^1 & (t'-1)^2 & \cdots & (t'-1)^{t'-1} \end{bmatrix},$$

$$Y = \begin{bmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,t'-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,t'-1} \\ & & \vdots & \\ c_{t'-1,0} & c_{t'-1,1} & \cdots & c_{t'-1,t'-1} \end{bmatrix},$$

$$D = \begin{bmatrix} \left\lfloor \frac{0}{0} \right\rfloor & \left\lfloor \frac{0}{1} \right\rfloor & \cdots & \left\lfloor \frac{0}{t'-1} \right\rfloor \\ \left\lfloor \frac{1}{0} \right\rfloor & \left\lfloor \frac{1}{1} \right\rfloor & \cdots & \left\lfloor \frac{1}{t'-1} \right\rfloor \\ & & \vdots & \\ \left\lfloor \frac{t'-1}{0} \right\rfloor & \left\lfloor \frac{t'-1}{1} \right\rfloor & \cdots & \left\lfloor \frac{t'-1}{t'-1} \right\rfloor \end{bmatrix},$$

$$Z = \begin{bmatrix} c'_{0,0} & c'_{0,1} & \cdots & c'_{0,t'-1} \\ c'_{1,0} & c'_{1,1} & \cdots & c'_{1,t'-1} \\ & & \vdots & \\ c'_{t'-1,0} & c'_{t'-1,1} & \cdots & c'_{t'-1,t'-1} \end{bmatrix}$$

and then we have

$$VY = D \ \text{ and } \ VZ = Y^\top,$$

so the coefficients of polynomials $g_j(x)$ can be precomputed from

$$Z = V^{-1}D^\top V^{-\top} \bmod t.$$

3.2.2 Our Improved Algorithm

Now we describe, given $[\![(a_0, a_1, \ldots, a_{N-1})]\!]$ and $[\![(d_0, d_1, \ldots, d_{N-1})]\!]$, how to compute

$$\left[\!\!\left[\left(\left\lfloor \frac{a_0}{d_0} \right\rfloor, \left\lfloor \frac{a_1}{d_1} \right\rfloor, \ldots, \left\lfloor \frac{a_{N-1}}{d_{N-1}} \right\rfloor\right)\right]\!\!\right].$$

First from the input ciphertexts, we compute the following t' ciphertexts of powers

$$[\![(a_0^i, a_1^i, \ldots, a_{N-1}^i)]\!],\ [\![(d_0^i, d_1^i, \ldots, d_{N-1}^i)]\!] \ \text{ where } 0 \le i \le t'-1$$

by Pows (Algorithm 1). Next, for each d_i, we want to obtain the coefficients of $f_{d_i}(x)$, i.e., $(c_{0,d_i}, c_{1,d_i}, \ldots, c_{t'-1,d_i})$. For that, we compute the j-th coefficients $(c_{j,d_0}, c_{j,d_1}, \ldots, c_{j,d_{N-1}})$ by polynomial evaluation of $g_j(x)$ for $0 \le j \le t'-1$ as

$$\begin{aligned}
[\![(c_{j,d_0}, c_{j,d_1}, \ldots, c_{j,d_{N-1}})]\!] \leftarrow\ & [(c'_{0,j}, c'_{0,j}, \ldots, c'_{0,j})] \\
& \oplus [(c'_{1,j}, c'_{1,j}, \ldots, c'_{1,j})] \otimes [\![(d_0, d_1, \ldots, d_{N-1})]\!] \\
& \oplus [(c'_{2,j}, c'_{2,j}, \ldots, c'_{2,j})] \otimes [\![(d_0^2, d_1^2, \ldots, d_{N-1}^2)]\!] \\
& \qquad\qquad \vdots \\
& \oplus [(c'_{t'-1,j}, c'_{t'-1,j}, \ldots, c'_{t'-1,j})] \otimes [\![(d_0^{t'-1}, d_1^{t'-1}, \ldots, d_{N-1}^{t'-1})]\!].
\end{aligned}$$

Finally we compute the encrypted division result by parallel polynomial evaluation of $f_{d_i}(x)$ as

$$\begin{aligned}
\left[\!\!\left[\left(\left\lfloor \frac{a_0}{d_0} \right\rfloor, \left\lfloor \frac{a_1}{d_1} \right\rfloor, \ldots, \left\lfloor \frac{a_{N-1}}{d_{N-1}} \right\rfloor\right)\right]\!\!\right] \leftarrow\ & [\![(c_{0,d_0}, c_{0,d_1}, \ldots, c_{0,d_{N-1}})]\!] \\
& \oplus [\![(c_{1,d_0}, c_{1,d_1}, \ldots, c_{1,d_{N-1}})]\!] \otimes [\![(a_0, a_1, \ldots, a_{N-1})]\!] \\
& \oplus [\![(c_{2,d_0}, c_{2,d_1}, \ldots, c_{2,d_{N-1}})]\!] \otimes [\![(a_0^2, a_1^2, \ldots, a_{N-1}^2)]\!] \\
& \qquad\qquad \vdots \\
& \oplus [\![(c_{t'-1,d_0}, c_{t'-1,d_1}, \ldots, c_{t'-1,d_{N-1}})]\!] \otimes [\![\left(a_0^{t'-1}, a_1^{t'-1}, \ldots, a_{N-1}^{t'-1}\right)]\!].
\end{aligned}$$

The prior method [OCHK18] requires t' polynomial evaluations for each equality check and division respectively, that is, $2t'$ times in total, while our method requires only t' polynomial evaluations, thus leading to better efficiency. We note that each polynomial evaluation includes t' plaintext-ciphertext multiplications and t' ciphertext-ciphertext additions. The run-times of one plaintext-ciphertext multiplication and ciphertext-ciphertext addition are small, but the total number of these required operations is quadratic in t', so as the input domain size t' becomes larger, the damage to the efficiency is non-negligible (see Table 2). When $t' = 2^{15}$, the estimate of the total computation time is about 59 days in our experimental environment (see §4.2) even if our improved algorithm is used. Therefore, we propose another method (§3.4) to compute an arbitrary bivariate

integer function in a reasonably efficient way even when the input domain size is $t' = 2^{15}$ at the price of allowing only a single slot to be used during the computation of bivariate functions.

3.3 Homomorphic Evaluation of Arbitrary Univariate Function

3.3.1 One-HotSlot

As a very simple but important building block, first we propose Algorithm 3 which we call One-HotSlot and it can be used to compute an arbitrary univariate function later. On input $[\![(a, a, \ldots, a)]\!]$ where $a \in \mathbb{Z}_N$, One-HotSlot computes a packed ciphertext in which only the a-th slot is 1 and all other slots are 0 with a zero-based index. We note that in Step 3 of Algorithm 3 all non-zero slot values are set to 1 by Fermat's little theorem since t is a prime.

Algorithm 3 One-HotSlot

Input: Packed ciphertext $[\![a]\!] = [\![(a, a, \ldots, a)]\!]$

1: $S \leftarrow [(0, 1, 2, \ldots, N-1)]$

2: $\ell \leftarrow [\![a]\!] \ominus S$
$= [\![\underbrace{(a, a-1, a-2, \ldots, 0, \ldots, a-N+1)}_{\text{only } a\text{-th slot is } 0}]\!]$

3: $m \leftarrow \ell^{t-1}$ // computed by repeated squaring modulo t
$= [\![(1, 1, \ldots, 0, 1, \ldots, 1)]\!]$

4: $n \leftarrow [(1, 1, \ldots, 1)] \ominus m$
$= [\![\underbrace{(0, 0, \ldots, 1, 0, \ldots, 0)}_{\text{only } a\text{-th slot is } 1}]\!]$

Output: n

3.3.2 Univariate Function Evaluation with One-HotSlot and EvalSum

By recalling the packing method in §2.2, we observe that if we take the sum of all the elements of the $k(\neq 0)$-th column of the matrix $W \in \mathbb{Z}_t^{N \times N}$, we obtain

$$\begin{aligned}
\sum_{i=0}^{N-1} \omega^{(2i+1)k} &\equiv \omega^k + (\omega^3)^k + (\omega^5)^k + (\omega^7)^k + \cdots + (\omega^{2N-1})^k \\
&\equiv \omega^k (1 + \omega^{2k} + \cdots + \omega^{2(N-1)k}) \\
&\equiv \omega^k (1 + \omega^{2k} + \cdots + (\omega^{2k})^{(N-1)}) \\
&\equiv \omega^k \cdot \frac{1 - (\omega^{2k})^N}{1 - \omega^{2k}} \\
&\equiv 0 \bmod t \quad (\because \omega^{2N} \equiv 1 \bmod t).
\end{aligned}$$

Based on the observation above, further we can derive the following useful fact regarding $[\![f(X)]\!] = [\![(a'_0, a'_1, \ldots, a'_{N-1})]\!]$:

$$\begin{aligned}\sum_{i=0}^{N-1} a'_i &\equiv N \cdot a_0 \\ &\quad + a_1 \cdot \{\omega + (\omega^3) + \cdots + (\omega^{2N-1})\} \\ &\quad + a_2 \cdot \{\omega^2 + (\omega^3)^2 + \cdots + (\omega^{2N-1})^2\} \\ &\quad \vdots \\ &\quad + a_{N-1} \cdot \{\omega^{N-1} + (\omega^3)^{N-1} + \cdots + (\omega^{2N-1})^{N-1}\} \\ &\equiv N \cdot a_0 \\ &\equiv N \cdot f(0) \bmod t \end{aligned} \tag{1}$$

In [IY18], an algorithm called ConstantTermExtract is proposed to compute $[\![f(0)]\!]$ from $[\![f(X)]\!]$, which is based on the idea from [CH18, A.1]. ConstantTermExtract can be realized by using Galois automorphisms and key switching. By combining the relation in Eq. (1) with ConstantTermExtract, we can construct EvalSum of Algorithm 4 which, on input the ciphertext encrypting $[f(X)] = [(a'_0, a'_1, \ldots, a'_{N-1})]$, returns the packed ciphertext consisting of the sums of all the slots $\sum_{i=0}^{N-1} a'_i$. We can construct EvalSum just by slightly modifying ConstantTermExtract in [IY18].

Algorithm 4 EvalSum

Input: Packed ciphertext $[\![f(X)]\!] = [\![(a'_0, a'_1, \ldots, a'_{N-1})]\!]$

1: $c \leftarrow \sigma_{N+1}([\![(a'_0, a'_1, \ldots, a'_{N-1})]\!])$
2: $c \leftarrow [\![(a'_0, a'_1, \ldots, a'_{N-1})]\!] \oplus c$
3: **for** $k = 1$ to $\log_2(N) - 1$ **do**
4: $\quad c' \leftarrow \sigma_{\frac{N}{2^k}+1}(c)$
5: $\quad c \leftarrow c \oplus c'$
6: **end for**

Output: c $(= [\![N \cdot f(0)]\!] = [\![\sum_{i=0}^{N-1} a'_i]\!] = [\![(\sum_{i=0}^{N-1} a'_i, \sum_{i=0}^{N-1} a'_i, \ldots, \sum_{i=0}^{N-1} a'_i)]\!])$

Here $\sigma_i(f(X))$ means the automorphism mapping for $i \in \mathbb{Z}^*_{2N}$. For example, if $N = 2^2$, given $f(X) = 1 + 2X + 3X^2 + 4X^3$, we have

$$\sigma_5(f(X)) = 1 + 2(X^5) + 3(X^5)^2 + 4(X^5)^3 = 1 - 2X + 3X^2 - 4X^3 \bmod X^N + 1.$$

The plaintext polynomial encrypted in the output ciphertext c of EvalSum consists of only a constant term $N \cdot f(0)$, so all the slots have the same value $\sum_{i=0}^{N-1} a'_i$ according to Eq. (1).

Now we propose Algorithm 5[13] to compute arbitrary univariate functions such as bit/digit decomposition, which is realized by the table lookup method combining One-HotSlot and EvalSum.

[13] The range of function f can actually be $\mathbb{Z}_t$.

Algorithm 5 Homomorphic Evaluation of Arbitrary Univariate Function

Input: Packed ciphertext $[\![a]\!] = [\![(a, a, \ldots, a)]\!]$ and packed plaintext $T = [(f(0), f(1), \ldots, f(N-1))]$ representing a lookup table of function $f\colon \mathbb{Z}_N \to \mathbb{Z}_N$

1: $\ell \leftarrow \mathsf{One\text{-}HotSlot}([\![a]\!])$
2: $m \leftarrow \ell \otimes T = \ell \otimes [(f(0), f(1), \ldots, f(N-1))]$
3: $n \leftarrow \mathsf{EvalSum}(m)$

Output: $n = [\![f(a)]\!]$

EvalSum performs automorphism mapping[14] and addition of ciphertexts $\log_2(N)$ times, and its computational cost is about 6 times as large as that of a ciphertext-ciphertext multiplication. In our experiment, it takes about one second to perform EvalSum once. Hence we try to minimize the number of EvalSum operations as well as ciphertext-ciphertext multiplication in our construction.

3.3.3 Set-Membership Predicate by Univariate Function

We can see that a set-membership predicate $f_S(x) = (x \in S) \in \{0, 1\}$ can be easily constructed by slightly modifying univariate function evaluation (Algorithm 5). I.e., if we use S of $f_S(x)$ in One-HotSlot (Algorithm 3) and $T = [(1, 1, \ldots, 1, 1)]\,(= [1])$ in Algorithm 5, then clearly this variant of Algorithm 5 returns $f_S(a)$.

3.3.4 Comparison Operation by Univariate Function

We can see that $\mathsf{One\text{-}HotSlot}([\![a]\!])$ (Algorithm 3) returns the all-zero vector $[\![(0, \ldots, 0)]\!]$ if $a \notin S(= \{0, 1, \ldots, N-1\})$ as mentioned in §3.3.3. Hence, given the two ciphertexts of $a, b \in \{0, 1, \ldots, N-1\}$, we can compute the ciphertext of the comparison result $(a \le b) \in \{0, 1\}$ just by inputting $[\![b - a]\!]$ and $T = [(1, 1, \ldots, 1, 1)]$ to Algorithm 5 where we note that the computation of $b - a$ is done modulo t and we have $-(N-1) \le b - a \le N - 1$. This way of computing $(a \le b)$ works correctly because we have $0 \le b - a \le N - 1$ if and only if $(a \le b) = 1$ and Algorithm 5 returns $[\![1]\!]$ if $0 \le b - a \le N - 1$ and $[\![0]\!]$ otherwise.

3.3.5 Simulating Bivariate Function by Univariate Function

Assuming that the input domain size is small, we can simulate a bivariate function just by computing a univariate function $\mathbb{Z}_N \to \mathbb{Z}_N$ using One-HotSlot. For example, suppose we compute the division $[\![\lfloor \frac{a}{d} \rfloor]\!]$ with inputs $[\![a]\!]$ and $[\![d]\!]$ where $a < 2^7$, $d < 2^8$ and the input domain size N of the univariate function is $N = 2^{15}$. In this case, first we compute $[\![a \times 2^8 + d]\!]$, and the division can be realized by preparing a lookup table T whose i-th slot is the division result $\lfloor \frac{a}{d} \rfloor$ when $i = a \cdot 2^8 + d$, i.e., $T = \left[\left(\left\lfloor \frac{0}{0} \right\rfloor, \left\lfloor \frac{0}{1} \right\rfloor, \ldots, \left\lfloor \frac{0}{2^8-1} \right\rfloor, \left\lfloor \frac{1}{0} \right\rfloor, \ldots, \left\lfloor \frac{2^7-1}{2^8-1} \right\rfloor\right)\right]$. We show this division algorithm (which we call SmallDivision) in Algorithm 6.

[14] An automorphism actually needs a key switching operation, but we omit it here because the detail of it is not essential in this work.

Algorithm 6 SmallDivision

Input: Packed ciphertexts $[\![a]\!] = [\![(a, a, \ldots, a)]\!]$, $[\![d]\!] = [\![(d, d, \ldots, d)]\!]$ where $a < 2^7$, $d < 2^8$

1: $T \leftarrow \left[\left(\left\lfloor \frac{0}{0} \right\rfloor, \left\lfloor \frac{0}{1} \right\rfloor, \ldots, \left\lfloor \frac{0}{2^8-1} \right\rfloor, \left\lfloor \frac{1}{0} \right\rfloor, \ldots, \left\lfloor \frac{2^7-1}{2^8-1} \right\rfloor\right)\right]$

2: $k \leftarrow [\![a]\!] \otimes 2^8 \oplus [\![d]\!]$

3: $\ell \leftarrow \mathsf{One\text{-}HotSlot}(k)$

4: $m \leftarrow \ell \otimes T$

5: $n \leftarrow \mathsf{EvalSum}(m)$

Output: n // corresponding to $[\![\lfloor \frac{a}{d} \rfloor]\!]$

This way of computing a bivariate function is simple, but if we compute a bivariate function $f\colon \mathbb{Z}_N \times \mathbb{Z}_N \to \mathbb{Z}_N$, we need to use the ciphertext space $\mathbb{Z}_q[X]/(X^{N^2}+1)$, which will be prohibitive when $N = 2^{15}$. In the next section, we overcome this issue by showing how to compute $f\colon \mathbb{Z}_N \times \mathbb{Z}_N \to \mathbb{Z}_N$ with the ciphertext space $\mathbb{Z}_q[X]/(X^N+1)$.

3.4 Homomorphic Evaluation of Arbitrary Bivariate Function

We show how to compute an arbitrary bivariate function $f(x, y)\colon \mathbb{Z}_N \times \mathbb{Z}_N \to \mathbb{Z}_N$[15] by combining the table lookup method using One-HotSlot with a parallel polynomial evaluation using the PS method.

First we define a polynomial $f_d(x) = f(x, d)$

$$f_d(x) = c_{0,d} + c_{1,d}x + c_{2,d}x^2 + \cdots + c_{N-1,d}x^{N-1} \bmod t$$

and the coefficients $c_{i,d}$ of $f_d(x)$ can be precomputed via polynomial interpolation[16]. Hence we can precompute all the coefficients of polynomials $f_0(x), \ldots, f_{d_i}(x), \ldots f_{N-1}(x)$ where $d = 0, 1, \ldots, d_i, \ldots, N-1$.

Next given the input packed ciphertexts $[\![a]\!] = [\![(a, a, \ldots, a)]\!]$[17], we want to compute

$$[\![f_0(a), \ldots, f_{d_i}(a), \ldots f_{N-1}(a)]\!]$$

by using the PS method (§2.4). For that, we use Pows and compute the following powers

$$([\![a]\!], [\![a^2]\!], \ldots, [\![a^{p-1}]\!]),\ ([\![a^p]\!], [\![(a^p)^2]\!], \ldots, [\![(a^p)^{s-1}]\!])$$

with multiplicative depth $\log(p)$ and $\log(ps)$ respectively where $N = ps$.

[15] The input domain size can be smaller than N, and the range can actually be $\mathbb{Z}_t$.

[16] As an alternative, we can use $g_a(x) = f(a, x)$ instead of $f_d(x)$ if the polynomial $g_a(x)$ has a simpler structure (such as an odd polynomial) than $f_d(x)$.

[17] If the input ciphertext is of form $[\![(0, \ldots, 0, a, 0, \ldots, 0)]\!]$, it can be transformed into the required form by applying EvalSum.

Now we can obtain $[\![f_0(a), \ldots, f_{d_i}(a), \ldots f_{N-1}(a)]\!]$ by computing

$$
\begin{aligned}
&[\![f_0(a), \ldots, f_{d_i}(a), \ldots f_{N-1}(a)]\!] \leftarrow \\
&\quad (c_0 + c_1[\![a]\!] + \cdots + c_{p-1}[\![a^{p-1}]\!]) \\
&\quad + \left(c_p + c_{p+1}[\![a]\!] + \cdots + c_{2p-1}[\![a^{p-1}]\!]\right) \times [\![a^p]\!] \\
&\quad + \left(c_{2p} + c_{2p+1}[\![a]\!] + \cdots + c_{3p-1}[\![a^{p-1}]\!]\right) \times [\![(a^p)^2]\!] \\
&\qquad\qquad\qquad \vdots \\
&\quad + \left(c_{(s-1)p} + c_{(s-1)p+1}[\![a]\!] + \cdots + c_{(s-1)p+p-1}[\![a^{p-1}]\!]\right) \times [\![(a^p)^{s-1}]\!]
\end{aligned}
$$

where each c_i $(0 \le i < N)$ is the following packed plaintext

$$
\begin{aligned}
c_0 &= \left[(c_{0,0},\, c_{0,1}, \ldots, c_{0,d}, \ldots, c_{0,N-1})\right] \\
c_1 &= \left[(c_{1,0},\, c_{1,1}, \ldots, c_{1,d}, \ldots, c_{1,N-1})\right] \\
&\vdots \\
c_{N-1} &= \left[(c_{N-1,0},\, c_{N-1,1}, \ldots, c_{N-1,d}, \ldots, c_{N-1,N-1})\right].
\end{aligned}
$$

Further by multiplying $[\![f_0(a), \ldots, f_{d_i}(a), \ldots f_{N-1}(a)]\!]$ with the packed ciphertext $\mathsf{One\text{-}HotSlot}([\![d]\!])$ in which only the d-th slot is 1 and all other slots are 0, we obtain the packed ciphertext in which the d-th slot contains the function output $f_d(a)\,(= f(a, d))$ and the other slots are set to 0. This can be viewed as a table lookup operation using a one-hot vector. Finally by applying $\mathsf{EvalSum}$, we obtain the final packed ciphertext in which all the slots contain the function output. The above procedure is given in Algorithm 7.

Algorithm 7 Homomorphic Evaluation of Arbitrary Bivariate Function

Input: Packed ciphertexts $[\![a]\!] = [\![(a,\, a, \ldots, a)]\!]$, $[\![d]\!] = [\![(d,\, d, \ldots, d)]\!]$

1: (Precomputations): Precompute coefficients $c_{i,d}$ of the polynomials $f_d(x)$ where $0 \le d < N$, and obtain the packed plaintexts c_i where $0 \le i < N$.
2: $C_a^{pow} \leftarrow \mathsf{Pows}([\![a]\!],\, p)$
3: $C_{a^p}^{pow} \leftarrow \mathsf{Pows}([\![a^p]\!],\, s)$ where $N = ps$
4: $S \leftarrow 0$
5: **for** $i = 0$ to s **do**
6: $\quad f \leftarrow 0$
7: $\quad$ **for** $j = 0$ to p **do**
8: $\quad\quad f \leftarrow f \oplus c_{i \times p + j} \otimes C_a^{pow}[j]$
9: $\quad$ **end for**
10: $\quad S \leftarrow S \oplus f \otimes C_{a^p}^{pow}[i]$
11: **end for**
12: $S \leftarrow S \otimes \mathsf{One\text{-}HotSlot}([\![d]\!])$
13: $r \leftarrow \mathsf{EvalSum}(S)$

Output: r // corresponding to the ciphertext of $f(a,\, d)$

3.5 Complexity Analysis

Here we summarize the complexities of (i) our algorithm in §3.2, (ii) Algorithm 7, and (iii) [OCHK18], and for ease of exposition, we refer to (i) as Proposal 1 and (ii) as Proposal 2. Table 2 shows the approximate numbers of operations required for one invocation of $f\colon \mathbb{Z}_{t'} \times \mathbb{Z}_{t'} \to \mathbb{Z}_{t'}$ in terms of plaintext-ciphertext multiplication (pt $\times$ ct), ciphertext-ciphertext addition (ct + ct), ciphertext-ciphertext multiplication (ct $\times$ ct), and MakePackedPlaintext. Here MakePackedPlaintext is an operation[18] that creates packed plaintexts used as coefficients in polynomial evaluations. As shown in Table 4, this operation is at least heavier than pt $\times$ ct in our experimental environment, so this operation is counted in the complexity analysis. In Proposal 2, t' needs to satisfy $t' \leq N$ where N is the degree defining $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, and the parameters p, s for the PS method need to satisfy $t' = p \times s$ and $p, s \approx \sqrt{t'}$, and t is the plaintext modulus.

We note that, in [OCHK18], Proposal 1, and Proposal 2, the degree of the polynomials to be evaluated is $t'-1$, and computing the powers $(\llbracket a^0 \rrbracket, \llbracket a \rrbracket, \llbracket a^2 \rrbracket, \ldots, \llbracket a^{t'-1} \rrbracket)$ with $\mathsf{Pows}(\llbracket a \rrbracket, t')$ (Algorithm 1) requires approximately t' multiplications (i.e., ct $\times$ ct). In [OCHK18] and Proposal 1, the polynomial evaluation (related to $\mathsf{ConstDiv}, \mathsf{ConstEq}$ in Algorithm 2) is done by computing the inner product between the powers and polynomial coefficients, which involves pt $\times$ ct and ct + ct. In Proposal 2, the polynomial evaluation is done with the PS method, and in addition, Proposal 2 involves $\mathsf{One\text{-}HotSlot}$ and $\mathsf{EvalSum}$. From these facts, we can derive the complexities in Table 2. When $t' = N = 2^{15}$, we can see that the term t'^2 in pt $\times$ ct, ct + ct, and MakePackedPlaintext of [OCHK18] and Proposal 1 becomes quite large, and this causes the computational bottleneck even if the complexities of pt $\times$ ct, ct + ct, and MakePackedPlaintext are smaller than that of ct $\times$ ct, thus yielding the efficiency gaps among the schemes.

We summarize the tradeoff between Proposals 1 and 2. In Proposal 1, the input domain size can be set flexibly without being limited to N, and the SIMD functionality can be available with an appropriate parameter setting, but the term t'^2 in the complexity affects the run-time severely as shown in Table 3. Meanwhile in Proposal 2, the input domain size is limited to[19] N which is also related to the ciphertext size and the SIMD functionality is not available during the computation of bivariate functions, but by removing the term t'^2 in the complexity, the increase of the run-time is much smaller compared with Proposal 1 as shown in Table 3.

3.6 Possible Extensions

3.6.1 Extending Input Domain Size

The input domain size of Algorithm 7 is N, and we can extend it such that the bivariate function $f(x, y)$ is of type $\mathbb{Z}_{2N} \times \mathbb{Z}_{2N} \to \mathbb{Z}_t$ with a relatively moderate overhead. Our idea is simple, that is, we separately deal with the cases where

[18] This operation is called MakePackedPlaintext in PALISADE [PAL20].
[19] This can be slightly extended to $2N$ and t as shown in §3.6.1 and §A.1.

Table 2. Complexity Comparison of Computing $f\colon \mathbb{Z}_{t'} \times \mathbb{Z}_{t'} \to \mathbb{Z}_{t'}$

	[OCHK18]	Proposal 1	Proposal 2
pt × ct	$2{t'}^2$	${t'}^2$	t'
ct + ct	$2{t'}^2 + t'$	${t'}^2 + t'$	$t' + \log_2(N) + s$
ct × ct	$3t'$	$3t'$	$\log_2(t-1) + p + 2s$
MakePackedPlaintext	$2{t'}^2$	${t'}^2$	t'

the second input is between 0 and $N-1$, and between N and $2N-1$. We modify Algorithm 7, and give Algorithm 8 for larger inputs.

Algorithm 8 Homomorphic Evaluation of Arbitrary Bivariate Function with Larger Inputs ($\mathbb{Z}_{2N} \times \mathbb{Z}_{2N} \to \mathbb{Z}_t$)

Input: Packed ciphertexts $[\![a]\!] = [\![(a, a, \ldots, a)]\!]$, $[\![d]\!] = [\![(d, d, \ldots, d)]\!]$

1: (Precomputations): Precompute coefficients $c_{i,d}$ of the polynomial $f_d(x)$ where $0 \leq d < N$ and coefficients $c'_{i,d}$ of $f_d(x)$ where $N \leq d < 2N$, and obtain the packed plaintexts c_i and c'_i where $0 \leq i < 2N$.
2: $C_a^{pow} \leftarrow \mathsf{Pows}([\![a]\!], p)$
3: $C_{a^p}^{pow} \leftarrow \mathsf{Pows}([\![a^p]\!], s)$ where $2N = ps$
4: $S \leftarrow 0,\ S' \leftarrow 0$
5: **for** $i = 0$ to s **do**
6: $\quad f \leftarrow 0,\ f' \leftarrow 0$
7: $\quad$ **for** $j = 0$ to p **do**
8: $\quad\quad f \leftarrow f \oplus c_{i\times p+j} \otimes C_a^{pow}[j]$
9: $\quad\quad f' \leftarrow f' \oplus c'_{i\times p+j} \otimes C_a^{pow}[j]$
10: $\quad$ **end for**
11: $\quad S \leftarrow S \oplus f \otimes C_{a^p}^{pow}[i]$
12: $\quad S' \leftarrow S' \oplus f' \otimes C_{a^p}^{pow}[i]$
13: **end for**
14: $r \leftarrow S \otimes \mathsf{One\text{-}HotSlot}([\![d]\!]) \oplus S' \otimes \mathsf{One\text{-}HotSlot}([\![d-N]\!])$
15: $r \leftarrow \mathsf{EvalSum}(r)$

Output: r // corresponding to the ciphertext of $f(a, d)$

$\mathsf{One\text{-}HotSlot}([\![d]\!])$ returns a one-hot packed ciphertext if $0 \leq d < N$, and $[\![0]\!]$ otherwise. Similarly $\mathsf{One\text{-}HotSlot}([\![d-N]\!])$ returns a one-hot packed ciphertext if $N \leq d < 2N$, and $[\![0]\!]$ otherwise[20]. Hence the correct value is selected depending on the second input d at Step 14 of Algorithm 8. By using the same technique, actually we can realize the bivariate function of type $\mathbb{Z}_t \times \mathbb{Z}_t \to \mathbb{Z}_t$, and in the

[20] We can see that $\mathsf{One\text{-}HotSlot}([\![d-N]\!])$ corresponds to invoking $\mathsf{One\text{-}HotSlot}([\![d]\!])$ with a packed plaintext $[(N, N+1, N+2, \ldots, 2N-1)]$ instead of $[(0, 1, 2, \ldots, N-1)]$ in Algorithm 3.

typical setting $t = 2N + 1$, we need three invocations of One-HotSlot at Step 14 of Algorithm 8 (see Algorithm 10) [21].

3.6.2 Applying Bootstrap Procedure

The limitation of our scheme is that it consumes a relatively large multiplicative depth, and it is roughly $\log_2(t-1)$ when t is a plaintext modulus. In our experiments with PALISADE's BFVrns, the maximal depth is 23 with the setting $(N, t) = (2^{15}, 2^{16} + 1)$ and the maximal depth is 42 with the setting $(N, t) = (2^{15}, 3 \times 2^{18} + 1)$. To have the more remaining depth, one possible solution is of course to apply the general bootstrap procedure [CH18], assuming that FHE applications we deal with do not require real-time response. Another possibility is to apply the more lightweight TFHE-style bootstrap procedure [CGGI20][22], assuming that the bootstrapped ciphertext has the plaintext in only one slot. The message encoding method of BFV is the same as TFHE, i.e., the encoding of $a \in \mathbb{Z}_N (\subset \mathbb{Z}_t)$ is $\lfloor \frac{q}{t} \cdot a \rceil \in \mathbb{Z}_q$, so in general the TFHE-style bootstrap procedure can be applied, but there is an issue we need to address. When the input domain size is $\mathbb{Z}_N$, we need to have a ciphertext space $\mathbb{Z}_q[X]/(X^{N'}+1)$ during the TFHE-style bootstrap procedure where $N' > N$ because N' needs to have "finer granularity" to tolerate and remove the noise, but this can damage the efficiency of bootstrapping. To keep $N' = N$, the plaintext in the bootstrapped ciphertext should not be so large, so we homomorphically decompose the plaintext into the base-d representation where $d \ll N$, and the ciphertext of each digit is bootstrapped to be combined later. This digit decomposition can be done simultaneously in a univariate/bivariate function f we originally want to compute. Now we briefly describe the procedure assuming the reader's familiarity with the TFHE-style bootstrap procedure.

1. Let $f(x, y) \colon \mathbb{Z}_N \times \mathbb{Z}_N \to \mathbb{Z}_N$ be the bivariate function we originally want to compute. Then we define functions $\{f_i(x, y)\}_{0 \le i \le \ell - 1}$ where $\ell = \lfloor \log_d(N-1) \rfloor + 1$ such that if $z = f(a, b)$ and the base-d representation of z is $(z_{\ell-1}, z_{\ell-2}, \ldots, z_i, \ldots, z_1, z_0)_d$, then

$$f_i(a, b) = N \cdot \overbrace{\underbrace{\left\lfloor \frac{1}{2} \cdot \frac{1}{d} \cdot \frac{t}{2} + \frac{1}{d} \cdot \frac{t}{2} \cdot z_i \right\rceil}_{< \frac{t}{2}}}^{z'_i} \bmod t.$$

   The reason why we have $f_i(a, b) = N \cdot z'_i \bmod t$ instead of $f_i(a, b) = z'_i$ comes from the technicality of letting z'_i correspond to the constant term as in Eq. (1). The reason why $z'_i < \frac{t}{2}$ is necessary is that the TFHE-style bootstrap procedure has the "negacyclicity" constraint. The value $f_i(a, b)$

[21] This technique is also applicable to Algorithm 5 to realize the univariate function of type $\mathbb{Z}_t \to \mathbb{Z}_t$ (see Algorithm 9).

[22] We note that this bootstrap procedure can also be used in the setting different from the torus as in [MP21,KS21].

is determined such that $z_i \in \{0, 1, \ldots, d-1\}$ is mapped to the center of the z_i-th block of width $\frac{1}{d} \cdot \frac{t}{2}$.

2. After computing f_i with Algorithm 7 but without[23] invoking $\mathsf{EvalSum}$ in Step 13, we obtain the ciphertext $\mathsf{ct}_{i,\mathrm{BFV}}$ encrypting a polynomial whose constant term is z'_i. Next by applying TFHE's $\mathsf{SampleExtract}$ to $\mathsf{ct}_{i,\mathrm{BFV}}$, we obtain the LWE ciphertext $\mathsf{ct}_{i,\mathrm{LWE}}$ whose plaintext is $\lfloor \frac{q}{t} \cdot z'_i \rceil + e$ where e is the noise to be removed.
3. To apply the TFHE-style bootstrap procedure called "functional bootstrapping" to $\mathsf{ct}_{i,\mathrm{LWE}}$ with TFHE's $\mathsf{BlindRotate}$, we use the following test polynomial[24] $v_i(X)$,

$$v_i(X) = \overbrace{\underbrace{v_{i,0} + v_{i,1}X + v_{i,2}X^2 + \cdots}^{\text{width } \frac{N}{d}}}^{\text{coef. } v_{i,k} = \lfloor \frac{q}{t} \cdot 0 \cdot d^i \rceil} + \overbrace{\underbrace{\cdots \qquad \cdots}_{\text{width } \frac{N}{d}}}^{\text{coef. } v_{i,k} = \lfloor \frac{q}{t} \cdot 1 \cdot d^i \rceil} + \cdots + \overbrace{\underbrace{+ v_{i,N-1}X^{N-1}}_{\text{width } \frac{N}{d}}}^{\text{coef. } v_{i,k} = \lfloor \frac{q}{t} \cdot (d-1) \cdot d^i \rceil}.$$

   As a result, we obtain the BFV ciphertext $\mathsf{ct}'_{i,\mathrm{BFV}}$ encrypting a polynomial whose constant term is $z_i \cdot d^i \in \mathbb{Z}_N (\subset \mathbb{Z}_t)$ (corresponding to $\lfloor \frac{q}{t} \cdot z_i \cdot d^i \rceil \in \mathbb{Z}_q$).
4. We compute the BFV ciphertext $\mathsf{ct}'_{\mathrm{BFV}} \leftarrow \sum_{i=0}^{\ell-1} \mathsf{ct}'_{i,\mathrm{BFV}}$ encrypting a polynomial whose constant term is the result of $f(x, y) \in \mathbb{Z}_N$.
5. We apply $\mathsf{SampleExtract}$ to $\mathsf{ct}'_{\mathrm{BFV}}$ and obtain the LWE ciphertext $\mathsf{ct}'_{\mathrm{LWE}}$ whose plaintext is the result of $f(x, y)$. Further by applying TFHE's TLWE-to-$\mathsf{T(R)LWE}$ algorithm [CGGI20] to $\mathsf{ct}'_{\mathrm{LWE}}$, we obtain the BFV ciphertext $\mathsf{ct}''_{\mathrm{BFV}}$ [25] encrypting a polynomial consisting of only a constant term corresponding to the result of $f(x, y)$.

Implementing the above procedure in PALISADE [PAL20] with the instantiated parameters can be a challenging task, and we leave it as future work.

3.6.3 Further Extending Input Domain Size

We show another approach to extending the input domain size from $\mathbb{Z}_t$ to $\mathbb{Z}_{t^2}$ in Appendix A by using two ciphertexts to encrypt one integer plaintext. More precisely we realize the homomorphic operations for addition, multiplication, subtraction, comparison, and arbitrary univariate functions in $\mathbb{Z}_{t^2}$ by employing our techniques for uni/bivariate function evaluation in $\mathbb{Z}_t$.

4 Implementation

We show the implementation results of (i) our algorithm in §3.2 (referred to as Proposal 1), (ii) Algorithm 7 (referred to as Proposal 2), and (iii) [OCHK18] where we adopt the division operation as a bivariate integer function.

[23] We avoid $\mathsf{EvalSum}$ for efficiency reason.

[24] This is also known as a test vector.

[25] As an alternative, we can obtain $\mathsf{ct}''_{\mathrm{BFV}}$ just by computing $N^{-1} \cdot \mathsf{EvalSum}(\mathsf{ct}'_{\mathrm{BFV}})$ because $\mathsf{EvalSum}$ can extract $[\![N \cdot f(0)]\!]$ from $[\![f(X)]\!]$.

4.1 Implementation Details

Our implementation was done with C++ using multi-threaded PALISADE [PAL20]. The PC used for the experiment was Ubuntu 20.04 OS with Ryzen 5 3600@3.6GHz CPU (6 cores, 12 threads) and 128 GB RAM. We note that the run-time required for precomputation and the run-time required to read the precomputation results from the file are not included. The FHE scheme we use is PALISADE's BFVrns, and the securityLevel variable is set to HEStd_128_classic as a security parameter. In the implementation of [OCHK18] without a packing method, for each bit length ℓ of input integers, i.e., for the input domain size 2^ℓ, the smallest prime greater than 2^ℓ can be used as a plaintext modulus t. On the other hand, since our scheme uses a packing method, the plaintext modulus is fixed as $t = 2N + 1$ where $N = 2^{15}$ in our implementation.

4.2 Experimental Result

Table 3 shows the results of our experiment where L_0, L_1 and L_2 are the level parameter required by [OCHK18], Proposal 1 and Proposal 2 respectively. The level parameter should be at least the multiplicative depth the underlying algorithm requires.

Table 4 shows the run-times of plaintext-ciphertext multiplication (pt × ct), ciphertext-ciphertext addition (ct + ct), ciphertext-ciphertext multiplication (ct × ct), and MakePackedPlaintext respectively where the plaintext modulus is $t = 2^{16} + 1$ and L is the level parameter.

Table 3. Run-Time (s) of Bivariate Function $f\colon \mathbb{Z}_{2^\ell} \times \mathbb{Z}_{2^\ell} \to \mathbb{Z}_{2^\ell}$

ℓ	L_0	L_1	L_2	[OCHK18]	Proposal 1	Proposal 2
3	4	4	17	0.65	0.61	10.74
4	5	5	17	1.75	1.27	11.00
5	6	6	17	6.44	3.76	11.74
6	7	7	17	18.89	10.96	12.56
7	8	8	17	69.62	36.35	14.52
8	9	9	17	268.33	131.82	16.01
9	10	10	17	1083.41	537.65	20.79
10	11	11	17	4236.46	2140.61	25.90
11	12	12	17	43703.7	21499.1	38.86
12	-	-	17	-	-	57.49
13	-	-	17	-	-	98.72
14	-	-	17	-	-	163.28
15	-	-	17	-	-	306.93

Table 4. Run-Time (ms) of Each Basic Operation

L	pt × ct	ct + ct	ct × ct	MakePackedPlaintext
4	0.30	0.67	17.35	0.57
5	0.33	0.77	17.21	0.62
6	0.31	0.75	19.17	0.63
7	0.33	0.86	20.67	0.63
8	0.67	0.63	23.56	0.69
9	0.30	0.98	22.69	0.72
10	0.57	0.62	24.95	0.81
11	0.44	0.68	25.66	0.76
12	0.53	1.63	58.73	1.64
13	0.43	2.14	66.28	1.79
14	0.47	1.81	67.20	1.80
15	0.49	2.35	74.98	1.91
16	0.47	2.35	74.82	1.91

From Table 3, we can see that Proposal 1 (our improved version of [OCHK18]) is faster than Proposal 2 when $\ell \leq 6$, [26] but Proposal 2 outperforms Proposal 1 when $\ell \geq 7$. Although Proposal 1 halves the number of polynomial evaluations compared to the prior work [OCHK18], the estimation[27] based on Tables 2 and 4 indicates that the run-time of Proposal 1 with $\ell = 15$ (i.e., input domain size $N = 2^{15}$) is about 58.7 days in our experimental environment and it also means that the run-time of the prior work [OCHK18] is about 117.3 days, which can be prohibitively inefficient. The estimated amortized run-time (corresponding to the run-time per slot) is 154.8 seconds in Proposal 1 when $\ell = 15$, and unless the parallel computation needs more than about 16523 slots, Proposal 2 can be more advantageous.

5 Conclusion

In this work, we improved the efficiency of the prior work [OCHK18] for homomorphically evaluating arbitrary bivariate integer functions. We decreased the number of polynomial evaluations, thereby halving the run-time of the prior work [OCHK18]. We also proposed another algorithm for homomorphically evaluating arbitrary bivariate integer functions with a relatively large input domain size

[26] The reason for this comes from the difference in the key generation part. Proposal 2 requires additional key generation for automorphism mappings (equivalent to PALISADE's EvalAutomorphismKeyGen). The automorphism keyGen always takes about 7.3 seconds in our experimental environment regardless of ℓ in Proposal 2, whereas it takes only 0.1 seconds in Proposal 1 and [OCHK18]. Hence Proposal 1 is finished during the key generation in Proposal 2 in these cases.

[27] In our experimental environment, the estimated run-times tend to be smaller than the real run-times.

($\mathbb{Z}_N \times \mathbb{Z}_N \to \mathbb{Z}_N$) by exploiting the packing technique instead of enabling the SIMD operation. Further we showed that the input domain size can be extended such that the bivariate function is of type $\mathbb{Z}_{2N} \times \mathbb{Z}_{2N} \to \mathbb{Z}_{2N}$ with a relatively moderate overhead. Investigating the viability of implementing the combination of our approach and the TFHE-style bootstrap procedure is left as future work.

Acknowledgments. We would like to thank Koji Nuida for informing us of a simple polynomial expression formula [KMNN15, Theorem 2] that can be used to compute multiplication carry over a finite prime field, which led to the more efficient construction in Appendix A.3. This work was supported in part by JSPS KAKENHI Grant Number 20K11807.

References

BGV12. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS*, pages 309–325. ACM, 2012. 2, 5

Bra12. Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *CRYPTO*, pages 868–886. Springer, 2012. 2

CGGI20. 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. 2, 18, 19

CH18. Hao Chen and Kyoohyung Han. Homomorphic lower digits removal and improved FHE bootstrapping. In *Eurocrypt*, pages 315–337. Springer, 2018. 3, 12, 18

CKK20. Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. Efficient homomorphic comparison methods with optimal complexity. In *Asiacrypt*, pages 221–256. Springer, 2020. 4

CKKS17. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Asiacrypt*, pages 409–437. Springer, 2017. 2, 5

CLOT21. Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In *Asiacrypt*, pages 670–699. Springer, 2021. 2

DM15. Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *Eurocrypt*, pages 617–640. Springer, 2015. 2

FV12. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. 2, 5

GBA21. 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. 2

Gen09. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178. ACM, 2009. 2

HS14. Shai Halevi and Victor Shoup. Algorithms in HElib. In *CRYPTO*, pages 554–571. Springer, 2014. 2

HS21. Shai Halevi and Victor Shoup. Bootstrapping for HElib. *Journal of Cryptology*, 34(1):1–44, 2021. 3

INZ21. Ilia Iliashenko, Christophe Negre, and Vincent Zucca. Integer functions suitable for homomorphic encryption over finite fields. In *WAHC*, pages 1–10. ACM, 2021. 4

IY18. Yu Ishimaki and Hayato Yamana. Non-interactive and fully output expressive private comparison. In *Indocrypt*, pages 355–374. Springer, 2018. 3, 12

IZ21. Ilia Iliashenko and Vincent Zucca. Faster homomorphic comparison operations for BGV and BFV. *PoPETs*, 2021(3):246–264, 2021. 4

KLLW16. Myungsun Kim, Hyung Tae Lee, San Ling, and Huaxiong Wang. On the efficiency of FHE-based private queries. *IEEE Transactions on Dependable and Secure Computing*, 15(2):357–363, 2016. 4

KMNN15. Shizuo Kaji, Toshiaki Maeno, Koji Nuida, and Yasuhide Numata. Polynomial expressions of carries in p-ary arithmetics. *arXiv preprint arXiv:1506.02742*, 2015. 22, 27

KMNN19. Shizuo Kaji, Toshiaki Maeno, Koji Nuida, and Yasuhide Numata. Polynomial expressions of p-ary auction functions. *Journal of Mathematical Cryptology*, 13(2):69–80, 2019. 3

KPZ21. Andrey Kim, Yuriy Polyakov, and Vincent Zucca. Revisiting homomorphic encryption schemes for finite fields. In *Asiacrypt*, pages 608–639. Springer, 2021. 2

KS21. Kamil Kluczniak and Leonard Schild. FDFB: Full domain functional bootstrapping towards practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2021/1135, 2021. 18

LZS18. Wen-jie Lu, Jun-Jie Zhou, and Jun Sakuma. Non-interactive and output expressive private comparison from homomorphic encryption. In *ASIACCS*, pages 67–74. ACM, 2018. 3

MMN22. Daisuke Maeda, Koki Morimura, and Takashi Nishide. Efficient homomorphic evaluation of arbitrary bivariate integer functions. In *Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC)*, pages 13–22. ACM, 2022. 1

MP21. Daniele Micciancio and Yuriy Polyakov. Bootstrapping in FHEW-like cryptosystems. In *WAHC*, pages 17–28. ACM, 2021. 18

NO07. Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *PKC*, pages 343–360. Springer, 2007. 26

OCHK18. Hiroki Okada, Carlos Cid, Seira Hidano, and Shinsaku Kiyomoto. Linear depth integer-wise homomorphic division. In *IFIP International Conference on Information Security Theory and Practice*, pages 91–106. Springer, 2018. 3, 4, 7, 8, 9, 10, 16, 17, 19, 20, 21

PAL20. PALISADE Lattice Cryptography Library (release 1.10.6). `https://palisade-crypto.org/`, December 2020. 2, 16, 19, 20

PS73. Michael S Paterson and Larry J Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing*, 2(1):60–66, 1973. 3, 4, 7

SEA22. Microsoft SEAL (release 4.0). `https://github.com/Microsoft/SEAL`, March 2022. Microsoft Research, Redmond, WA. 2

SV14. Nigel P Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Designs, codes and cryptography*, 71(1):57–81, 2014. 3, 5

TLW+20. Benjamin Hong Meng Tan, Hyung Tae Lee, Huaxiong Wang, Shuqin Ren, and Khin Mi Mi Aung. Efficient private comparison queries over encrypted databases using fully homomorphic encryption with finite fields. *IEEE Transactions on Dependable and Secure Computing*, 18(6):2861–2874, 2020. 4

A Extending Input Domain Size from $\mathbb{Z}_t$ to $\mathbb{Z}_{t^2}$

Here we extend the input domain size from $\mathbb{Z}_t$ to $\mathbb{Z}_{t^2}$ by using two ciphertexts to encrypt one integer plaintext. Now let $A = a_1 \times t + a_0 = (a_1, a_0)_t \in \mathbb{Z}_{t^2}$ and $B = b_1 \times t + b_0 = (b_1, b_0)_t \in \mathbb{Z}_{t^2}$ in the base-t representation where $a_0, a_1, b_0, b_1 \in \mathbb{Z}_t$, and to encrypt plaintexts A, B, we have four (BFV) ciphertexts encrypting a_0, a_1, b_0, b_1. Then we consider how to simulate homomorphic operations in $\mathbb{Z}_{t^2}$ by homomorphic operations in $\mathbb{Z}_t$ with the ciphertexts of a_0, a_1, b_0, b_1. In this appendix, we assume the typical setting $(N, t) = (2^{15}, 2N+1) = (2^{15}, 2^{16}+1)$, and give several optimizations using the properties of this setting.

A.1 Extending Algorithms 5 and 8

Before describing the homomorphic operations in $\mathbb{Z}_{t^2}$, first we extend Algorithm 5 for a function of type $\mathbb{Z}_N \to \mathbb{Z}_N$ such that the evaluated function is of type $\mathbb{Z}_t \to \mathbb{Z}_t$ (Algorithm 9). Similarly we extend Algorithm 8 for a function of type $\mathbb{Z}_{2N} \to \mathbb{Z}_{2N}$ such that the evaluated function is of type $\mathbb{Z}_t \times \mathbb{Z}_t \to \mathbb{Z}_t$ (Algorithm 10).

Algorithm 9 Univariate Function Evaluation for $f\colon \mathbb{Z}_t \to \mathbb{Z}_t$

Input: $[\![a]\!] = [\![(a, a, \ldots, a)]\!]$, 3 packed plaintexts $T = [(f(0), f(1), \ldots, f(N-1))]$, $T' = [(f(N), f(N+1), \ldots, f(2N-1))]$, $T'' = [(f(\underbrace{t-1}_{2N}), 0, \ldots, 0)]$

1: $\ell \leftarrow \mathsf{One\text{-}HotSlot}([\![a]\!])$
2: $\ell' \leftarrow \mathsf{One\text{-}HotSlot}([\![a - N]\!])$
3: $m \leftarrow \ell \otimes T \oplus \ell' \otimes T' \oplus \underbrace{\{1 \ominus ([\![a]\!] \ominus 2N)^{t-1}\} \otimes T''}_{\text{handling case of } a = t-1\,(=2N)}$
4: $n \leftarrow \mathsf{EvalSum}(m)$

Output: $n = [\![f(a)]\!]$

Algorithm 10 Bivariate Function Evaluation for $f\colon \mathbb{Z}_t \times \mathbb{Z}_t \to \mathbb{Z}_t$

Input: $[\![a]\!] = [\![(a, a, \ldots, a)]\!]$, $[\![b]\!] = [\![(b, b, \ldots, b)]\!]$

1: (Precomputations): Precompute coefficients $c_{i,b}$ of the polynomial $f_b(x)\,(= f(x, b))$ where $0 \le b < N$, coefficients $c'_{i,b}$ of $f_b(x)$ where $N \le b < 2N$, and coefficients $c''_{i,2N}$ where $b = t - 1\,(= 2N)$, and obtain the packed plaintexts c_i, c'_i, c''_i where $0 \le i < t$.
2: $C_a^{pow} \leftarrow \mathsf{Pows}([\![a]\!],\, p)$
3: $C_{a^p}^{pow} \leftarrow \mathsf{Pows}([\![a^p]\!],\, s)$ where $p,\, s \approx \sqrt{t}$ & $t \le ps$
4: $S \leftarrow 0,\ S' \leftarrow 0,\ S'' \leftarrow 0$
5: **for** $i = 0$ to s **do**
6: $\quad f \leftarrow 0,\ f' \leftarrow 0,\ f'' \leftarrow 0$
7: $\quad$ **for** $j = 0$ to p **do**
8: $\quad\quad f \leftarrow f \oplus c_{i\times p+j} \otimes C_a^{pow}[j]$
9: $\quad\quad f' \leftarrow f' \oplus c'_{i\times p+j} \otimes C_a^{pow}[j]$
10: $\quad\quad f'' \leftarrow f'' \oplus c''_{i\times p+j} \otimes C_a^{pow}[j]$
11: $\quad$ **end for**
12: $\quad S \leftarrow S \oplus f \otimes C_{a^p}^{pow}[i]$
13: $\quad S' \leftarrow S' \oplus f' \otimes C_{a^p}^{pow}[i]$
14: $\quad S'' \leftarrow S'' \oplus f'' \otimes C_{a^p}^{pow}[i]$
15: **end for**
16: $r \leftarrow S \otimes \mathsf{One\text{-}HotSlot}([\![b]\!]) \oplus S' \otimes \mathsf{One\text{-}HotSlot}([\![b - N]\!]) \oplus \underbrace{S'' \otimes \{1 \ominus ([\![b]\!] \ominus 2N)^{t-1}\} \otimes [(1, 0, \ldots, 0)]}_{\text{handling case of } b = t-1\,(=2N)}$
17: $r \leftarrow \mathsf{EvalSum}(r)$

Output: r // corresponding to the ciphertext of $f(a, b)$

In Algorithm 10, $f_b(x)$ is a function that returns $f(a, b)$ for the input a. Since $t = 2N + 1$ and the number of slots in one ciphertext is N, we need three intermediate ciphertexts S, S', and S'' to handle the cases of $0 \le b < N$, $N \le b < 2N$, and $b = t - 1\,(= 2N)$ separately, and these cases can be processed in parallel.

A.2 Addition

We compute the ciphertexts of two digits of $A + B \bmod t^2 = (s_1, s_2)_t$ in the base-t representation. This addition can be represented in the following column addition (we note that actually a_0, a_1, b_0, $b_1 \in \mathbb{Z}_t$ are ciphertexts):

$$\begin{array}{cccc} & a_1 & & a_0 \\ + & b_1 & & b_0 \\ \hline * & \underbrace{\left\{\begin{array}{l}\mathsf{carry}(a_0,\, b_0)+\\ a_1 + b_1 \bmod t\end{array}\right\}}_{s_1} & & \underbrace{(a_0 + b_0 \bmod t)}_{s_0} \end{array}$$

Table 5. Truth Table for $\delta = (a_0 \geq b')$

$\alpha = \left(a_0 \leq \frac{t-1}{2}\right)$	$\beta = \left(b' \leq \frac{t-1}{2}\right)$	$\gamma = \left((a_0 - b' \bmod t) \leq \frac{t-1}{2}\right)$	$\delta = (a_0 \geq b')$
1	0	*	0
0	1	*	1
0	0	0	0
0	0	1	1
1	1	0	0
1	1	1	1

Here we can ignore the $*$ part because the computation is done modulo t^2, and $(a_0 + b_0 \bmod t)$ and $(a_1 + b_1 \bmod t)$ can be computed simply by homomorphic addition in $\mathbb{Z}_t$. Next we consider how to compute the addition carry $\mathsf{carry}(a_0, b_0) \in \{0, 1\}$. This carry can be computed as $(a_0 \geq b')$ where $b' = t - b_0 \, (= -b_0 \bmod t)$ by a bivariate function evaluation of $\mathbb{Z}_t \times \mathbb{Z}_t \to \mathbb{Z}_t$. However, we show that the computation of this carry can be decomposed into more lightweight univariate function evaluations as follows. First we define the following bits α, β, γ,

$$\alpha = \left(a_0 \leq \frac{t-1}{2}\right),\ \beta = \left(b' \leq \frac{t-1}{2}\right),\ \gamma = \left((a_0 - b' \bmod t) \leq \frac{t-1}{2}\right) \in \{0, 1\}.$$

By employing the truth table from [NO07] shown in Table 5, we observe that $\delta = (a_0 \geq b')$ can be computed as

$$\begin{aligned}\delta &= \bar{\alpha}\beta \vee \bar{\alpha}\bar{\beta}\gamma \vee \alpha\beta\gamma \\ &= (1-\alpha)\beta + (1-\alpha)(1-\beta)\gamma + \alpha\beta\gamma \\ &= \beta - \alpha\beta + \gamma + \gamma(2\alpha\beta - \beta - \alpha),\end{aligned}$$

and note that α, β, γ can be computed (in parallel) by univariate function evaluations of $\mathbb{Z}_t \to \mathbb{Z}_t$ in Algorithm 9. Further in the typical setting where $t = 2N+1$, we have

$$\alpha = \left(a_0 \leq \frac{t-1}{2}\right) = \left(0 \leq a_0 \leq \frac{t-1}{2}\right) = (0 \leq a_0 \leq N),$$

and $\alpha = (0 \leq a_0 \leq N) \in \{0, 1\}$ can be computed as

$$\alpha = (0 \leq a_0 \leq N) = \neg(N+1 \leq a_0 \leq t-1) = 1 - (N+1 \leq a_0 \leq \underbrace{t-1}_{2N}).$$

Hence actually α can be computed from $(N + 1 \leq a_0 \leq t - 1)$, which can be viewed as a set-membership predicate[28] and computed by Algorithm 5 (as mentioned in §3.3.3) that is more lightweight than Algorithm 9. We can also compute β, γ similarly to α by Algorithm 5 (rather than Algorithm 9).

[28] We note that the number of elements in the set of the set-membership predicate is N here, so we can use (a variant of) Algorithm 5 instead of Algorithm 9.

A.3 Multiplication

We compute the ciphertexts of two digits of $A \times B \bmod t^2 = (s_1, s_0)_t$ in the base-t representation. This multiplication can be represented in the following column multiplication:

$$
\begin{array}{cccc}
 & & a_1 & a_0 \\
\times & & b_1 & b_0 \\
\hline
 & & \mathsf{mcarry}(a_0, b_0) & (a_0 \times b_0 \bmod t) \\
 & * & (a_1 \times b_0 \bmod t) & \\
 & * & (a_0 \times b_1 \bmod t) & \\
* & * & & \\
\hline
* & * & \underbrace{\left\{\begin{array}{c}\mathsf{mcarry}(a_0, b_0) \\ +a_1 \times b_0 \\ +a_0 \times b_1 \bmod t\end{array}\right\}}_{s_1} & \underbrace{(a_0 \times b_0 \bmod t)}_{s_0}
\end{array}
$$

Here we can ignore the $*$ part, and $(a_0 \times b_0 \bmod t)$, $(a_1 \times b_0 \bmod t)$ and $(a_0 \times b_1 \bmod t)$ can be computed (in parallel) simply by homomorphic multiplication in $\mathbb{Z}_t$. Next we consider how to compute the multiplication carry $\mathsf{mcarry}(a_0, b_0) \in \mathbb{Z}_t$. This carry can be computed by a bivariate function evaluation of $\mathbb{Z}_t \times \mathbb{Z}_t \to \mathbb{Z}_t$ in Algorithm 10. However, again we show that the computation of this carry can be decomposed into more lightweight univariate function evaluations thanks to a polynomial $\Psi(x)$ given in [KMNN15]. In Theorem 2 of [KMNN15], it was proved that[29] for an odd prime t,

$$\mathsf{mcarry}(a_0, b_0) \equiv a_0 b_0 \times (\Psi(a_0 b_0) - \Psi(a_0) - \Psi(b_0) + \Psi(1)) \bmod t \qquad (2)$$

where $\Psi(x)$ is a degree-$(t-2)$ polynomial determined solely by t and can be represented concretely with the Bernoulli numbers.

Since $\Psi(x)$ can be viewed as a univariate function $\mathbb{Z}_t \to \mathbb{Z}_t$, we can use Algorithm 9 to compute $\Psi(a_0 b_0)$, $\Psi(a_0)$, and $\Psi(b_0)$ (in parallel) in Eq. (2)[30].

Further we can notice that $\mathsf{mcarry}(a_0, b_0) = 0$ if $a_0 = 0$ or $b_0 = 0$ because Eq. (2) has a factor $a_0 b_0$. Hence in computing the intermediate result $(\Psi(a_0 b_0) - \Psi(a_0) - \Psi(b_0) + \Psi(1))$ in Eq. (2), we may assume that $a_0 \neq 0$ and $b_0 \neq 0$ because even if $a_0 = 0$ or $b_0 = 0$, we have $\mathsf{mcarry}(a_0, b_0) = 0$ correctly by multiplying the intermediate result by $a_0 b_0$ last. Thus actually we can use a slightly more lightweight variant of Algorithm 9 to compute $\Psi(x)$ in which we need only two

[29] In [KMNN15, Theorem 2], a more general multivariate version of mcarry is given, but the bivariate version suffices for our purpose.

[30] As a small optimization, the invocations of $\mathsf{EvalSum}(\cdot)$ (Step 4 of Algorithm 9) to compute $\Psi(a_0 b_0)$, $\Psi(a_0)$, and $\Psi(b_0)$ can be merged into one.

packed plaintexts $T = [(\Psi(1), \Psi(2), \ldots, \Psi(N))]$ and $T' = [(\Psi(N+1), \Psi(N+2), \ldots, \Psi(\underbrace{t-1}_{2N}))]$ and can go without T''[31].

A.4 Comparison

We compute one (BFV) ciphertext representing the comparison result $(A < B) \in \{0, 1\}$. We note that with the plaintext space $\mathbb{Z}_{t^2}$, the plaintext integers can be viewed as both unsigned numbers in $[0,\, t^2 - 1]$ and signed numbers in $[-\frac{t^2-1}{2},\, \frac{t^2-1}{2}]$. We handle both cases.

A.4.1 Case of Unsigned Number Encoding

In this encoding, we let $\mathbb{Z}_{t^2} = \{0, 1, \ldots, t^2 - 1\}$, and then we can compute $(A < B)$ simply by the digit-by-digit comparison as

$$(A < B) = (a_1 == b_1) \times (a_0 < b_0) + (a_1 < b_1), \tag{3}$$

where $(a_1 < b_1)$ and $(a_0 < b_0)$ can be computed (in parallel) similarly to $\delta = (a_0 \geq b')$ computed in §A.2[32], and $(a_1 == b_1)$ can be computed as $1 - (a_1 - b_1)^{t-1} \bmod t$ with repeated squaring.

A.4.2 Case of Signed Number Encoding

In this encoding, $\{0, 1, \ldots, t^2 - 1\}$ is viewed as $\{0,\, 1, \ldots, \frac{t^2-1}{2},\, -\frac{t^2-1}{2},\, -\frac{t^2-1}{2} + 1, \ldots, -2,\, -1\}$. Here we can take the approach similar to that in §A.2. We note that if A and B have the same sign, the final output is $(A < B)$ computed just by Eq. (3), but if A and B have the different signs, the final output should be $(B \leq \frac{t^2-1}{2})$ (i.e., $(A < B) = 1$ if and only if B is non-negative). Hence first we compute $\lambda = (A \leq \frac{t^2-1}{2})$, $\theta = (B \leq \frac{t^2-1}{2})$ and $(A < B)$ (in parallel) by using Eq. (3) where[33]

$$\lambda = \left(A \leq \frac{t^2-1}{2}\right) = 1 - \left(\frac{t^2-1}{2} < A\right) \in \{0, 1\}$$
$$\theta = \left(B \leq \frac{t^2-1}{2}\right) = 1 - \left(\frac{t^2-1}{2} < B\right) \in \{0, 1\}$$
$$\frac{t^2-1}{2} = Nt + N = (N,\, N)_t.$$

[31] Accordingly we use $\mathsf{One\text{-}HotSlot}(\llbracket a - 1 \rrbracket)$ and $\mathsf{One\text{-}HotSlot}(\llbracket a - 1 - N \rrbracket)$ in Steps 1 and 2 of Algorithm 9 respectively.

[32] We note that, e.g., $(a_1 < b_1) = 1 - (a_1 \geq b_1)$.

[33] Since $\frac{t^2-1}{2}$ is a known value, we can compute, e.g., $(\frac{t^2-1}{2} < A)$ as

$$\left(\frac{t^2-1}{2} < A\right) = (N == a_1) \times (N < a_0) + (N < a_1)$$
$$= (N == a_1) \times (N + 1 \leq a_0 \leq t - 1) + (N + 1 \leq a_1 \leq t - 1).$$

by simplifying Eq. (3) with (a variant of) Algorithm 5 as in §A.2.

Then we can compute the final output $(A < B)_{\mathsf{final}} \in \{0, 1\}$ by

$$\begin{aligned}(A < B)_{\mathsf{final}} &= \underbrace{(\lambda == \theta)}_{A,\, B \text{ have same sign}} \times (A < B) + (\lambda \mathrel{!}= \theta) \times \theta \\ &= (1 - (\lambda - \theta)^2) \times (A < B) + (\lambda - \theta)^2 \times \theta \\ &= (1 - \lambda + 2\lambda\theta - \theta) \times (A < B) - \lambda\theta + \theta \quad (\because\ \lambda,\, \theta \in \{0, 1\}).\end{aligned}$$

A.5 Subtraction

We compute the ciphertexts of two digits of $A - B \bmod t^2 = (s_1,\, s_0)_t$ in the base-t representation. Given the ciphertext of B, it is usually easy to obtain the ciphretext of $-B$ in FHE, but in our setting of $\mathbb{Z}_{t^2}$, we need an additional task of obtaining $t^2 - B$ before computing $A - B \bmod t^2$. If $b_0 \neq 0$, $t^2 - B$ can be computed by column subtraction with $t^2 = (t-1) \times t + t$ as

$$\begin{array}{rcc} & t-1 & t \\ - & b_1 & b_0 \\ \hline & (-1 - b_1 \bmod t) & (-b_0 \bmod t), \end{array}$$

and if $b_0 = 0$, with $t^2 = t \times t + 0$ as

$$\begin{array}{rcc} & t & 0 \\ - & b_1 & 0 \\ \hline & (-b_1 \bmod t) & 0. \end{array}$$

As a result, we observe $t^2 - B = (-b_1 - b_0^{t-1},\, -b_0)_t$ (where $b_0^{t-1} \in \{0, 1\}$), and can compute

$$(s_1,\, s_0)_t = (A - B \bmod t^2) = (A + t^2 - B) = (a_1,\, a_0)_t + (-b_1 - b_0^{t-1},\, -b_0)_t \bmod t^2$$

by using homomorphic (column) addition in §A.2.

A.6 Univariate Function Evaluation in $\mathbb{Z}_{t^2}$

We can compute an arbitrary univariate function $f\colon \mathbb{Z}_{t^2} \to \mathbb{Z}_{t^2}$ on $A = (a_1,\, a_0)_t$ by performing bivariate function evaluation of $\mathbb{Z}_t \times \mathbb{Z}_t \to \mathbb{Z}_t$ in Algorithm 10 twice (in parallel) to obtain $(s_1,\, s_0)_t = f(A)$.

A.7 Implementation Results

In Table 6, we show the implementation results[34] of EvalAdd′ in §A.2, EvalMult′ in §A.3[35], EvalUnsignedComp in §A.4.1, EvalSignedComp in §A.4.2, and EvalSub′ in §A.5. The experimental environment here is the same as the one in §4, and L is the multiplicative depth necessary for each operation.

Table 6. Run-Time (s) of Extended Operations in $\mathbb{Z}_{t^2}$

Operation	L	Time (s)
EvalAdd′	18	14.57
EvalMult′	18	17.76
EvalUnsignedComp	19	26.42
EvalSignedComp	20	49.15
EvalSub′	18	15.96

[34] Although there are several parts that can be computed in parallel, the computations are done sequentially as a baseline.

[35] As in §4, the run-time required for precomputation of $\Psi(x)$ and the run-time required to read the precomputation results from the file are not included.