Journal of Information Processing Vol.31 288–298 (May 2023)

[DOI: 10.2197/ipsjjip.31.288]

Regular Paper

Accelerating Polynomial Evaluation for Integer-wise Homomorphic Comparison and Division

Koki Morimura[1,a)] Daisuke Maeda[1,b)] Takashi Nishide[1,c)]

Received: December 15, 2022, Accepted: February 2, 2023

Abstract: Fully homomorphic encryption (FHE) is a promising tool for privacy-preserving applications, and it enables us to perform homomorphic addition and multiplication on FHE ciphertexts without decrypting them. FHE has two types: one supporting the exact computation and the other supporting the approximate computation. Further the FHE schemes supporting the exact computation have two types, bit-wise FHE, which encrypts a plaintext bit by bit, and integer-wise FHE, which encrypts a plaintext as an integer. Both types of FHE are important depending on the types of computation we need to execute securely. In this work, we focus on integer-wise FHE, and propose improved methods for integer-wise homomorphic comparison and division operations. For a comparison operation, we improve on the work of Iliashenko and Zucca (PoPETs'21) whose complexity is $O(p)$ homomorphic multiplications, and achieve the complexity $O(\sqrt{p})$ where p is a plaintext modulus. For a division operation, as opposed to the work of Okada et al. (WISTP'18), we propose a simple method to reduce the processing time by introducing an equality function based on Fermat's little theorem without changing the multiplicative depth, and show the analysis of why this approach can achieve better efficiency in detail. In our homomorphic division, the number of interpolated polynomials is reduced by half, thus also achieving the reduction of the processing time of precomputations and the number of polynomials to be stored. We also implement our improved methods in HElib, which is one of popular FHE libraries using the BGV encryption. As a result, we show that, e.g., in the plaintext space $\mathbb{Z}_{257}$, our homomorphic comparison with the Paterson-Stockmeyer method is faster by a factor of about 14.5 compared with Iliashenko and Zucca (PoPETs'21) and our homomorphic division is faster by a factor of about 1.45 compared with Okada et al. (WISTP'18).

Keywords: multiparty computation, fully homomorphic encryption, FHE, HElib, homomorphic comparison, homomorphic division, polynomial interpolation

1. Introduction

1.1 Background

In recent years, cloud computing has become popular, but protecting the data outsourced to the cloud appropriately is still a challenge. Therefore, an effective method for operation of confidential information is becoming increasingly important. Fully homomorphic encryption (FHE) was introduced by Gentry [10] in 2009. Since then, many improvements have been proposed to diversify computable functions and security assumptions of FHE, and it has attracted much attention as one of promising tools for secure data outsourcing. In FHE, it is possible to compute both homomorphic addition and multiplication on FHE ciphertexts, and theoretically it means that arbitrary operations can be performed on encrypted data without decrypting the ciphertexts. However, there is still much room for improving the efficiency of performing homomorphic comparison/division operations and other nonlinear functions on FHE ciphertexts. As a result, FHE applications may settle for using only homomorphic addition and multiplication with approximate computation results instead of using inefficient methods for nonlinear functions such as comparison/division. Thus, the reduction of the processing time of homomorphic comparison/division operations will lead to increasing the number of scenes where FHE applications can be deployed.

1.2 Related Work

There are two types of FHE, one supporting the exact computation and the other supporting the approximate computation (CKKS [5]). In CKKS, the plaintext space is the real/complex numbers, and the approximate computation on fixed-point values is supported. Further, the FHE schemes supporting the exact computation have two types, bit-wise FHE (FHEW [7] and TFHE [6]) and integer-wise FHE (BGV [3] and BFV [9]) [*1]. In the bit-wise FHE, the plaintext space is $\mathbb{Z}_2$, and it is possible to compute comparison and division operations using Boolean circuits [4]. In the integer-wise FHE or CKKS, the ciphertexts can be simply added or multiplied. On the other hand, in the bit-wise FHE, the addition of two binary FHE ciphertexts needs carry processing and multiplications, so it can be a performance bottleneck. Therefore, although depending on the types of com-

[1] Unibersity of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
[a)] morimura.tkb@gmail.com
[b)] s2120546@s.tsukuba.ac.jp
[c)] nishide@risk.tsukuba.ac.jp

A preliminary version of this work appeared in Ref. [18]. In this extended version, we give an additional theoretical performance analysis of homomorphic comparison algorithms of ours and Iliashenko and Zucca's [15] in the setting without the limited input range in Section 3, newly implement both our algorithm and Iliashenko and Zucca's algorithm, and experimentally show the advantage of our homomorphic comparison in Section 5.1 and Table 2.

[*1] More exactly, the integer-wise FHE supports modular arithmetic over a finite field.

putation, integer-wise FHE schemes or CKKS can be advantageous in privacy-preserving machine learning algorithms [1], [2], [12], [16]. However, in the integer-wise FHE/CKKS, homomorphic comparison and division operations can be another performance bottleneck, so the privacy-preserving algorithms using integer-wise FHE/CKKS tend to avoid homomorphic comparison/division operations. The techniques used for computing nonlinear functions in CKKS are different from those of the integer-wise FHE because of the difference between the computations over finite fields and real/complex numbers, and in this work, we focus on the exact integer computation using BGV/BFV [3], [9].

The comparison operation in the integer-wise FHE was first proposed in Ref. [19] in 2017, and the division operation in Ref. [21] in 2018. The comparison algorithm in Ref. [19] is performed by evaluating an appropriate interpolated polynomial and multiplications, and it is possible to compute homomorphic comparison with the multiplicative depth of $O(\log p)$ in the plaintext space $\mathbb{Z}_p$. However, in this method, the size of the plaintext space has a large influence on the performance of the computation, which greatly limits the plaintext space $\mathbb{Z}_p$ that can be used for efficient computation. Another recent comparison algorithm by Iliashenko and Zucca [15] in 2021 can handle large inputs by representing the integers in the base-q representation, encoding the integers as the elements in the extension field $\mathbb{F}_{q^d}$, and comparing the corresponding vectors of coefficients in lexicographical order. Although [15] can handle larger inputs, the method and the special encoding in Ref. [15] are designed just for performing the comparison operation, so their homomorphic comparison using the special encoding cannot be seamlessly combined with normal homomorphic additions and multiplications in the mixed computation. However, their homomorphic comparison without the special encoding is still useful and we will focus on their approaches using univariate/bivariate polynomial evaluation for homomorphic comparison later.

The division algorithm in Okada et al. [21] is performed similarly to the comparison operation with polynomial evaluation and multiplications, and it is possible to compute division with the multiplicative depth of $O(\log p)$ in the plaintext space $\mathbb{Z}_p$. However, in this method, again the size of the plaintext space has a large influence on the performance of the computation, which greatly limits the plaintext space $\mathbb{Z}_p$ that can be used for efficient computation. Another recent division algorithm by Okada et al. [22] in 2020 uses the technique called functional bootstrapping from TFHE [6], but the implementation results in Ref. [22] show that only 4-bit inputs are handled and that the decryption error probability becomes non-negligible (e.g., 30%) with larger inputs.

1.3 Our Contributions

Focusing on the exact integer computation, for homomorphic comparison, we propose an algorithm which improves on the previous homomorphic comparison by Iliashenko and Zucca [15] [*2]. In Ref. [15], an approach using univariate polynomial evaluation is taken to realize homomorphic comparison with a complexity of $O(\sqrt{p})$ homomorphic multiplications where p is a plaintext modulus, and further to handle larger inputs, another approach using bivariate polynomial evaluation is taken, but with a worse complexity of $O(p)$ homomorphic multiplications. In this work, we show how to improve on Ref. [15] whose complexity is $O(p)$ homomorphic multiplications and achieve the complexity $O(\sqrt{p})$ only using univariate polynomial evaluation.

We implement both our proposed algorithm and Ref. [15] on HElib, and show that, e.g., in the plaintext space $\mathbb{Z}_{257}$, our homomorphic comparison is faster by a factor of about 14.5 compared with Ref. [15] without increasing the multiplicative depth. For homomorphic division, we propose an algorithm which improves on the previous homomorphic division by Okada et al. [21] by reducing the number of polynomial evaluations. We implement both our proposed algorithm and Ref. [21] on HElib, and show that, e.g., in the plaintext space $\mathbb{Z}_{257}$, our homomorphic division is faster by a factor of about 1.45 compared with Ref. [21] without increasing the multiplicative depth.

2. Preliminaries

2.1 Notation

In the FHE construction, a ring R is used, whose elements are written in lower case here; 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 denote a mod q, with coefficients range $(-\frac{q}{2}, \frac{q}{2}]$. In the BGV scheme, we use the quotient polynomial ring $R_q = \mathbb{Z}_q[x]/\Phi_m(x)$, where q is an odd positive number and $\Phi_m(x)$ is the m-th cyclotomic polynomial. A FHE ciphertext whose plaintext is a is denoted as C_a.

We denote the logarithm to base 2 as $\log(\cdot)$. We denote vectors in bold. The notation $\boldsymbol{v}_i$ refers to the i^{th} element of $\boldsymbol{v}$ where $i \geq 1$, while the inner product of two vectors $\boldsymbol{u}, \boldsymbol{v} \in R^n$ is denoted as $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \sum_{i=1}^{n} \boldsymbol{u}_i \cdot \boldsymbol{v}_i$. We write $s \xleftarrow{U} S$ to denote the process of sampling s uniformly at random over S. We write $e \leftarrow \chi$ to denote the process of sampling e according to the probability distribution χ over S.

2.2 Fully Homomorphic Encryption (FHE)

Fully homomorphic encryption allows arbitrary operations such as addition and multiplication to be computed while the data are encrypted. Traditional hohomorphic encryption include RSA encryption [26] and Elgamal encryption [8] which have multiplicative homomorphism for ciphertexts and Paillier encryption [23] which has additive homomorphism for ciphertexts. Fully Homomorphic Encryption (FHE) is an encryption scheme that has both additive and multiplicative homomorphism. The first construction of FHE [10] was given by Gentry in 2009.

In FHE, noise is added to the ciphertext to maintain the security of the encryption. This noise increases with each iteration of addition or multiplication. In particular, multiplications between ciphertexts increase the noise in ciphertexts more than addition. If this noise exceeds a threshold, it is impossible to decrypt the ciphertexts correctly. For this reason, a noise reduction method

[*2] Here we improve on the homomorphic comparison algorithm without the special encoding in Ref. [15], and this comparison algorithm can be combined with homomorphic additions/multiplications to realize general integer computation.

called bootstrapping was proposed by Gentry [11], but the number of bootstrapping operations should be kept as small as possible because the complexity of bootstrapping is large.

2.3 BGV Scheme

An FHE scheme is a public-key encryption scheme that includes both addition and multiplication operations on ciphertexts such that: $\mathsf{Dec}(C_a + C_b) = a + b, \mathsf{Dec}(C_a \times C_b) = a \times b$. The BGV scheme [3] is one of the integer-wise FHE schemes and it is implemented in the FHE library HElib [13], which is widely used in the research of FHE applications. The security of the BGV scheme is based on the standard assumptions of the Learning with Error (LWE) problem [25] or Ring-LWE (RLWE) problem [17].

The tuple of basic algorithms $\mathsf{E} = (\mathsf{E.Setup}, \mathsf{E.SecretKeyGen}, \mathsf{E.PublicKeyGen}, \mathsf{E.Enc}, \mathsf{E.Dec})$ of the BGV scheme based on RLWE is 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 odd modulus $q = q(\lambda)$. We define the polynomial ring $R = \mathbb{Z}_q[x]/\Phi_m(x)$. Set a plaintext modulus p that is relatively prime to q, where the plaintext space is given by $R_p = \mathbb{Z}_p[x]/\Phi_m(x)$. Set noise distributions χ_{key}, χ_{err} over ring R. Output $\mathsf{params} = (R, m, p, q, \chi_{key}, \chi_{err})$.
- $\mathsf{SecretKeyGen}(\mathsf{params})$: Sample $\boldsymbol{s} \leftarrow \chi_{key}$. Output the secret key $sk = \boldsymbol{s}$.
- $\mathsf{PublicKeyGen}(\mathsf{params}, sk)$: Take as input the secret key $sk = \boldsymbol{s}$ and params. Sample $\boldsymbol{a} \xleftarrow{U} R_q$ and $\boldsymbol{e} \leftarrow \chi_{err}$. Set $\boldsymbol{b} := [\boldsymbol{sa} + p\boldsymbol{e}]_q \in R_q$, and the public key as

$$\boldsymbol{pk} = (\boldsymbol{pk}_0, \boldsymbol{pk}_1) := (\boldsymbol{b}, -\boldsymbol{a}) \in R_q^2$$

- $\mathsf{Enc}(\mathsf{params}, pk, m)$: To encrypt a message $\boldsymbol{m} \in R_p$ [*3], sample $\boldsymbol{u} \leftarrow \chi_{key}$ and $\boldsymbol{e}_0, \boldsymbol{e}_1 \leftarrow \chi_{err}$, and output the ciphertext as

$$\boldsymbol{c} := (\boldsymbol{c}_0, \boldsymbol{c}_1) = ([\boldsymbol{m} + \boldsymbol{u} \cdot \boldsymbol{pk}_0 + p\boldsymbol{e}_0]_q, [\boldsymbol{u} \cdot \boldsymbol{pk}_1 + p\boldsymbol{e}_1]_q).$$

- $\mathsf{Dec}(\mathsf{params}, sk, \boldsymbol{c})$: Output the message $\boldsymbol{m} := [[\boldsymbol{c}_0 + \boldsymbol{c}_1\boldsymbol{s}]_q]_p$.
  The decryption works because

$$\begin{aligned}\boldsymbol{m} &:= [[\boldsymbol{c}_0 + \boldsymbol{c}_1\boldsymbol{s}]_q]_p \\ &= [[\boldsymbol{m} + \boldsymbol{u} \cdot \boldsymbol{pk}_0 + p\boldsymbol{e}_0 + (\boldsymbol{u} \cdot \boldsymbol{pk}_1 + p\boldsymbol{e}_1)\boldsymbol{s}]_q]_p \\ &= [[\boldsymbol{m} + \boldsymbol{u}(\boldsymbol{as} + p\boldsymbol{e}) + p\boldsymbol{e}_0 + (-\boldsymbol{ua} + p\boldsymbol{e}_1)\boldsymbol{s}]_q]_p \\ &= [[\boldsymbol{m} + p(\boldsymbol{ue} + \boldsymbol{e}_0 + \boldsymbol{e}_1\boldsymbol{s})]_q]_p \\ &= \boldsymbol{m}.\end{aligned}$$

2.4 Bit-wise and Integer-wise FHE

Fully homomorphic encryption (FHE) supporting the exact computation is classified into two types: bit-wise FHE that encrypts a plaintext bit by bit and integer-wise FHE that encrypts a plaintext as integers. In the bit-wise FHE, Boolean circuits can be used to perform operations. Addition based on the Boolean circuit requires a carry processing, and it can be disadvantageous since multiplication between FHE ciphertexts is required to perform such addition. However, it is possible to perform relatively efficient operations such as comparison and division operations using Boolean circuits.

On the other hand, integer-wise FHE can compute addition and multiplication efficiently, but there is room for improvement in comparison and division operations because the complexity is large.

2.5 HElib

HElib [13] is a software library that implements the BGV scheme in C++. HElib is based on the number theory library NTL [28]. In addition to the basic BGV scheme, HElib also supports SIMD operation proposed by Smart and Vercauteren [27]. The SIMD operation enables to pack multiple plaintexts into a single element of R_p with the Chinese Remainder Theorem; it also enables parallel component-wise computation of the plaintexts in the SIMD "slots". Hence it produces a much better amortised performance, due to parallelization.

HElib has an interface for the "constant" evaluation, where addition or multiplication with a plaintext is performed on the ciphertext. In this work, the addition and multiplication between ciphertexts are denoted as FHE.Add and FHE.Mult respectively, and the addition and multiplication between a ciphertext and plaintext are denoted as FHE.addConst and FHE.multConst respectively [*4]. These evaluations with constants are efficient when the addend or multiplier is not an encrypted value. In particular, a multiplication with a constant is quite efficient compared to the multiplication between ciphertexts; the multiplication with a constant does not increase the dimension of the resultant ciphertext, and we do not need to perform "Key Switching" after this operation.

2.6 Polynomial Interpolation

In Ref. [19], an integer-wise homomorphic comparison algorithm was proposed for the first time. In Ref. [21], an integer-wise homomorphic division algorithm was proposed. These algorithms are based on polynomial interpolation over a prime field. Polynomial interpolation with prime p is a process of constructing a polynomial $f(x)$ of degree at most n which satisfies $y_i \equiv f(x_i) \bmod p$, $i \in \{0, 1, ..., n\}$, where $(n+1)$ data points $\{x_i, y_i\}$ are given such that $x_i \neq x_j$ when $i \neq j$. We can compute the polynomial $f(x)$ by

$$f(x) \equiv \sum_{i=0}^{n} \left(\prod_{\substack{0 \le j \le n, \\ j \neq i}} \frac{x - x_j}{x_i - x_j} \right) \cdot y_i \bmod p. \tag{1}$$

The algorithms Comp (Algorithm 1), HalfCheck (Algorithm 2), ConstDiv (Algorithm 6), and ConstEq (Algorithm 7) are constructed based on the evaluation of interpolated polynomials over a prime field, as discussed in Sections 3, 4.

2.7 Paterson-Stockmeyer Method

To evaluate a polynomial of degree $(n-1)$ obtained by poly-

[*3] In our implementation, a plaintext is a polynomial consisting of only a constant term.

[*4] We will also use, e.g., + instead of FHE.Add when it is clear from the context.

nomial interpolation with n data points by using homomorphic multiplication, we need to compute powers $\{c^1, c^2, \ldots, c^{n-1}\}$ of a ciphertext c. Therefore, the complexity of evaluating a polynomial with the powers is $O(n)$ multiplications if we use a naive method. The Paterson-Stockmeyer method (PS-method) [24] can reduce the complexity to $O(\sqrt{n})$.

In the PS-method, given a polynomial $f(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, first we transform the polynomial $f(x)$ to

$$\begin{aligned} f(x) &= a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \\ &= a_{km-1}x^{km-1} + \cdots + a_1 x + a_0 \\ &= (a_{km-1}x^{k-1} + a_{km-2}x^{k-2} + \cdots + a_{km-k}) \times (x^k)^{m-1} \\ &\quad + (a_{k(m-1)-1}x^{k-1} + a_{k(m-1)-2}x^{k-2} + \cdots + a_{k(m-1)-k}) \\ &\quad \times (x^k)^{m-2} \qquad (2) \\ &\quad \cdots \\ &\quad + (a_{2k-1}x^{k-1} + a_{2k-2}x^{k-2} + \cdots + a_k) \times x^k \\ &\quad + (a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_0). \end{aligned}$$

where $n = k \times m$ and typically k, m are set to be $\approx \sqrt{n}$. Next we evaluate Eq. (2) by computing $\{x, x^2, \ldots, x^{k-1}\}$ and $\{x^k, (x^k)^2, \ldots, (x^k)^{m-1}\}$. Thus by applying Eq. (2) in the polynomial evaluation, the number of multiplications between variable x can decrease from $O(n)$ to $O(\sqrt{n})$ without increasing the multiplicative depth.

However, when the PS-method is used for evaluating many different polynomials with the same value for variable x, the total number of multiplications may be larger compared with the case where the PS-method is *not* used and the powers of x, $\{x, x^2, \ldots, x^{n-1}\}$ are computed in advance. It is because evaluating Eq. (2) with the PS-method includes $O(m)$ multiplications between the ciphertexts even if we are given $\{x, x^2, \ldots, x^{k-1}\}$ and $\{x^k, (x^k)^2, \ldots, (x^k)^{m-1}\}$ and this $O(m)$ multiplications needed for each polynomial evaluation can invalidate the advantage of the PS-method if we are evaluating many different polynomials with the same value for variable x. Therefore, we note that it may be more efficient *not* to use the PS-method in evaluating many different polynomials.

2.8 Fermat's Little Theorem

Our algorithm uses the Fermat's little theorem in FermatEq (Algorithm 9). If p is a primer and $a \not\equiv 0 \bmod p$, then the theorem states that $a^{p-1} \equiv 1 \bmod p$. We note that given a ciphertext of plaintext a, computing the ciphertext of plaintext a^{p-1} can be done by homomorphic multiplications with repeated squaring where the multiplicative depth is $\log(p-1)$.

2.9 About Complexity of Operation

In FHE, homomorphic multiplications have a much more computational complexity than the other operations (homomorphic addition, constant multiplication, and constant addition). Therefore, the number of homomorphic multiplications has the dominant impact on the overall processing time. In this work, we mainly focus on the number of required homomorphic multiplications to measure the complexity of algorithms. However, the experimental results in Section 5 measure the processing time of the whole process rather than the number of homomorphic multiplications.

3. Integer-wise Homomorphic Comparison

3.1 Previous Work of Homomorphic Comparison with Limited Input Range

In Ref. [19], an integer-wise homomorphic comparison algorithm was proposed for the first time (Algorithm 1). This algorithm uses the univariate polynomial evaluation to determine the sign (positive or negative) for the comparison operation. In Algorithm 1, $C_{(a \geq b)}$ is a ciphertext of plaintext $(a \geq b)$ (as mentioned in Section 2.1) where $(a \geq b)$ is 1 if $a \geq b$ is true, and 0 otherwise. Here the plaintext space $\mathbb{Z}_p$ is $[-\lfloor \frac{p}{2} \rfloor, \lfloor \frac{p}{2} \rfloor]$ with prime p, but the input plaintexts a, b need to satisfy $0 \leq a, b \leq \lfloor \frac{p}{2} \rfloor$.

This algorithm converts performing a comparison operation into polynomial evaluation, which can be computed by using only homomorphic multiplications and additions. To compute the powers of the ciphertext C_{a-b} given C_a and C_b, it is necessary to perform $O(p)$ homomorphic multiplications when the plaintext space is $\mathbb{Z}_p$.

Iliashenko and Zucca [15] further improved this approach of univariate polynomial evaluation by noting that the structure of the polynomial $f(x)$ in Algorithm 1 is similar to that of an odd polynomial. Because the coefficient of x^i of an odd polynomial is a zero if i is even, we have only to compute the odd powers of C_{a-b} in Algorithm 1, thus leading to better efficiency. Iliashenko and Zucca showed that the complexity of Algorithm 1 can be $O(\sqrt{p})$ homomorphic multiplications in comibation with the PS-method.

3.2 Previous Work of Homomorphic Comparison without Limited Input Range

The homomorphic comparison based on univariate polynomial evaluation in Refs. [15], [19] still has the limitation in the sense that the input plaintexts a, b need to satisfy $0 \leq a, b \leq \lfloor \frac{p}{2} \rfloor$ although the plaintext space of the underlying FHE is $\mathbb{Z}_p$. Therefore, Iliashenko and Zucca [15] proposed another homomorphic comparison algorithm without the limited input range. That is, the input range of plaintexts can be $[0, p-1]$ with the plaintext space $\mathbb{Z}_p$ [*5].

In Refs. [14], [15], it was proved that the homomorphic com-

Algorithm 1 Comp(C_a, C_b)

Require: Ciphertexts: C_a, C_b where $0 \leq a, b \leq \lfloor \frac{p}{2} \rfloor$

Ensure: $C_{(a \geq b)}$

1: (Precomputation): Using polynomial interpolation, compute polynomial $f(x)$ mod p of degree $p-1$ that satisfies

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

2: Homomorphically compute $C_{a-b} \leftarrow C_a - C_b$

3: Output $C_{(a \geq b)} \leftarrow f(C_{a-b})$

[*5] We note that with the plaintext space $\mathbb{Z}_p$, the plaintext integers can be viewed as both unsigned numbers in $[0, p-1]$ and signed numbers in $[-\lfloor \frac{p}{2} \rfloor, \lfloor \frac{p}{2} \rfloor]$. In our homomorphic comparison, we assume that the input integers are interpreted to belong to the range $[0, p-1]$ as in Ref. [15] although the case of $[-\lfloor \frac{p}{2} \rfloor, \lfloor \frac{p}{2} \rfloor]$ can also be handled in a similar way.

Algorithm 2 HalfCheck(C_a)

Require: Ciphertexts: C_a where $0 \le a \le p-1$

Ensure: $C_{(a \le \lfloor p/2 \rfloor)}$

1: (Precomputation): Using polynomial interpolation, compute polynomial $f(x)$ mod p of degree $p-1$ that satisfies

$$f(x) = \begin{cases} -1 & (\text{if } x = 1, 2, \ldots, \left\lfloor \frac{p}{2} \right\rfloor) \\ 0 & (\text{if } x = 0) \\ 1 & (\text{if } x = \left\lfloor \frac{p}{2} \right\rfloor + 1, \left\lfloor \frac{p}{2} \right\rfloor + 2, \ldots, p-1) \end{cases}$$

We note that polynomial $f(x)$ is an odd polynomial for which the coefficient of x^i is a zero if i is even.

2: $C_{\mathsf{subhalf}} = f(C_a)$

3: (Precomputation): Using polynomial interpolation for data points $(-1, 1)$, $(0, 1)$, $(1, 0)$, find polynomial $g(x)$ mod p that satisfies

$$g(x) = \begin{cases} 1 & (\text{if } x = -1, 0) \\ 0 & (\text{if } x = 1) \end{cases}$$

(and actually we have $g(x) = -(x-1)(x+2)/2 \bmod p$).

4: Output $C_{(a \le \lfloor p/2 \rfloor)} \leftarrow g(C_{\mathsf{subhalf}})$

parison without the limited input range can be realized by the following bivariate polynomial $\mathsf{BiComp}(X, Y)$ returning $(X < Y) \in \{0, 1\}$,

$$\mathsf{BiComp}(X, Y) = Y \cdot (X - Y) \cdot (X + 1) \cdot f(X, Y) \bmod p,$$

where $0 \le X, Y \le p-1$ and

$$f(X, Y) = \sum_{i=0}^{(p-3)/2} f_i(X) \cdot Z^i \ \ (\text{with } Z = Y \cdot (X - Y)), \qquad (3)$$

and $f_i(X)$'s [*6] are also polynomials of X of degree $p - 3 - 2i$. When evaluating the polynomial $f(X, Y)$, $Y \cdot (X - Y)$ is viewed as Z and $f(X, Y)$ is evaluated as a polynomial of Z, thereby decreasing the processing time of the homomorphic comparison. Each coefficient of $f(X, Y)$ viewed as a polynomial of Z is computed as a polynomial of X respectively. As analyzed in Refs. [14], [15], given the ciphertexts of X, Y, homomorphically evaluating $\mathsf{BiComp}(X, Y)$ requires $O(p)$ homomorphic multiplications because computing the powers of X already requires $O(p)$ homomorphic multiplications.

3.3 Our Homomorphic Comparison Algorithm

Iliashenko and Zucca [15] removed the input range limitation, but at the price of a worse complexity $O(p)$. Here we show that the homomorphic comparison without the limited input range can be achieved more efficiently than Ref. [15] with a complexity $O(\sqrt{p})$. To this end, first we prepare Algorithm 2 (HalfCheck(C_a)) that checks whether an input a satisfies $a \le \left\lfloor \frac{p}{2} \right\rfloor$ without decrypting ciphertext C_a by using a univariate polynomial. Further by combining HalfCheck(C_a) with the idea using a truth table in Ref. [20], we construct our homomorphic comparison algorithm without the limited input range.

3.3.1 Subroutine HalfCheck

We describe our subroutine HalfCheck(C_a). HalfCheck(C_a) takes the ciphertext C_a of plaintext $a \in \mathbb{Z}_p$ as input and outputs C_1 if plaintext a satisfies $a \le \left\lfloor \frac{p}{2} \right\rfloor$ and C_0 otherwise. Our HalfCheck(C_a) first finds a polynomial $f(x)$ mod p such that

[*6] The details of how to compute the coefficients of $f_i(X)$'s are described in the proof of Theorem 12 of Ref. [14].

Table 1 Truth table for $(a \ge b) \in \{0, 1\}$ where $0 \le a, b \le p-1$.

$\alpha = (a \le \lfloor \frac{p}{2} \rfloor)$	$\beta = (b \le \lfloor \frac{p}{2} \rfloor)$	$\gamma = (a - b \bmod p \le \lfloor \frac{p}{2} \rfloor)$	$\delta = (a \ge b)$
1	0	*	0
0	1	*	1
0	0	0	0
0	0	1	1
1	1	0	0
1	1	1	1

$$f(x) = \begin{cases} -1 & (\text{if } x = 1, 2, \ldots, \left\lfloor \frac{p}{2} \right\rfloor) \\ 0 & (\text{if } x = 0) \\ 1 & (\text{if } x = \left\lfloor \frac{p}{2} \right\rfloor + 1, \left\lfloor \frac{p}{2} \right\rfloor + 2, \ldots, p-1) \end{cases}$$

by using polynomial interpolation. In the plaintext space $\mathbb{Z}_p$, the range $[\left\lfloor \frac{p}{2} \right\rfloor + 1, p - 1]$ can also be viewed as the range $[-\left\lfloor \frac{p}{2} \right\rfloor, -1]$ [*7]. Hence similarly to the observation already made in Ref. [15], the polynomial $f(x)$ in HalfCheck(C_a) is an odd polynomial, i.e., $f(x) = -f(-x) \bmod p$, so we have only to compute the odd powers of C_a, leading to better efficiency. By substituting C_a for x in $f(x)$, the ciphertext C_a can be converted into $C_{\mathsf{subhalf}} \in \{C_{-1}, C_0, C_1\}$. Next HalfCheck($C_a$) finds a quadratic polynomial $g(x)$ that transforms the input $\{-1, 0, 1\}$ to 1 (if input $= -1, 0$) or 0 (if input $= 1$). Finally we can obtain the ciphertext $C_{(a \le \lfloor p/2 \rfloor)}$ by substituting C_{subhalf} for x in $g(x)$ to obtain $\{C_0, C_1\}$ in HalfCheck(C_a).

3.3.2 Removing Input Range Limitation Efficiently

Here we show how to efficiently handle the case where the inputs to the homomorphic comparison satisfy $0 \le a, b \le p-1$ (rather than $0 \le a, b \le \left\lfloor \frac{p}{2} \right\rfloor$) by combining HalfCheck (Algorithm 2) with a truth table (**Table 1**) constructed for the secure multiparty comparison protocol of Nishide and Ohta [20].

First we note that computing $C_\alpha \leftarrow \mathsf{HalfCheck}(C_a)$ (where $\alpha \in \{0, 1\}$) can be viewed as

$$C_\alpha = \begin{cases} C_1 & (\text{if } a = 0, 1, \ldots, \left\lfloor \frac{p}{2} \right\rfloor) \\ C_0 & (\text{if } a = \left\lfloor \frac{p}{2} \right\rfloor + 1, \left\lfloor \frac{p}{2} \right\rfloor + 2, \ldots, p-1). \end{cases}$$

Then given C_a, C_b, we compute

$$C_\alpha = C_{(a \le \lfloor p/2 \rfloor)} \leftarrow \mathsf{HalfCheck}(C_a)$$
$$C_\beta = C_{(b \le \lfloor p/2 \rfloor} \leftarrow \mathsf{HalfCheck}(C_b)$$
$$C_\gamma = C_{(a-b \bmod p \le \lfloor p/2 \rfloor)} \leftarrow \mathsf{HalfCheck}(C_{a-b}).$$

Our goal here is to compute $C_\delta = C_{(a \ge b)}$ from C_α, C_β, C_γ. From the truth table of Table 1, we can see that the following holds for $\alpha, \beta, \gamma, \delta \in \{0, 1\}$ as observed in Ref. [20].

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

Hence we can compute C_α, C_β, C_γ with three invocations of HalfCheck, and $C_\delta = C_{(a \ge b)}$ from C_α, C_β, C_γ with two homomorphic multiplications as

[*7] As a toy example, when $p = 7$, the integers $\{0, 1, 2, 3, 4, 5, 6\}$ can also be viewed as $\{0, 1, 2, 3, -3, -2, -1\}$ because $4 \equiv -3$, $5 \equiv -2$, $6 \equiv -1 \bmod 7$.

Algorithm 3 Extended_Comp(C_a, C_b)

Require: Ciphertexts: C_a, C_b where $0 \le a, b \le p-1$
Ensure: $C_{(a \ge b)}$

1: $C_\alpha \leftarrow \mathsf{HalfCheck}(C_a)$
2: $C_\beta \leftarrow \mathsf{HalfCheck}(C_b)$
3: $C_{a-b} \leftarrow C_a - C_b$
4: $C_\gamma \leftarrow \mathsf{HalfCheck}(C_{a-b})$
5: $C_{\alpha\beta} \leftarrow \mathsf{FHE.Mult}(C_\alpha, C_\beta)$
6: $C_{2\alpha\beta-\beta-\alpha} \leftarrow 2 \times C_{\alpha\beta} - C_\beta - C_\alpha$
7: $C_{\gamma(2\alpha\beta-\beta-\alpha)} \leftarrow \mathsf{FHE.Mult}(C_\gamma, C_{2\alpha\beta-\beta-\alpha})$
8: $C_{(a \ge b)} \leftarrow C_\beta - C_{\alpha\beta} + C_\gamma + C_{\gamma(2\alpha\beta-\beta-\alpha)}$

$$C_{\alpha\beta} \leftarrow C_\alpha \times C_\beta$$
$$C_{2\alpha\beta-\beta-\alpha} \leftarrow 2 \times C_{\alpha\beta} - C_\beta - C_\alpha$$
$$C_{\gamma(2\alpha\beta-\beta-\alpha)} \leftarrow C_\gamma \times C_{2\alpha\beta-\beta-\alpha}$$
$$C_\delta \leftarrow C_\beta - C_{\alpha\beta} + C_\gamma + C_{\gamma(2\alpha\beta-\beta-\alpha)}.$$

We summarize this procedure Extended_Comp(C_a, C_b) in Algorithm 3. Because our Extended_Comp uses only univariate polynomial evaluation, the complexity of Extended_Comp can be $O(\sqrt{p})$ homomorphic multiplications in combination with the PS-method while the approach in Ref. [15] resorts to more costly bivariate polynomial evaluation, resulting in a complexity $O(p)$ even in combination with the PS-method.

4. Integer-wise Homomorphic Division

4.1 Previous Work of Homomorphic Division

In Ref. [21], an integer-wise homomorphic division algorithm [*8] was proposed for the first time (Algorithm 4), and is constructed based on the following subroutines:

- (Algorithm 5) Pows(C_a): Compute powers of ciphertext C_a.
- (Algorithm 6) ConstDiv($\boldsymbol{C}_a^{\mathsf{pow}}$, y): Integer-wise division of the ciphertext C_a by a public divisor y
- (Algorithm 7) ConstEq($\boldsymbol{C}_d^{\mathsf{pow}}$, y): Integer-wise equality check of the ciphertext C_d with a public input y

Div(C_a, C_d) homomorphically computes the ciphertext of a quotient $\lfloor a/d \rfloor$ denoted as $C_{\lfloor a/d \rfloor}$, from the ciphertexts of $a, d \in \mathbb{Z}_p$ denoted as C_a, C_d. Pows(C_a) homomorphically computes the ciphertexts of the powers $\{a, a^2, a^3, \ldots, a^{p-1}\}$, from the ciphertext of $a \in \mathbb{Z}_p$ denoted as C_a. ConstDiv($\boldsymbol{C}_a^{\mathsf{pow}}$, 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 divisor $y \in \mathbb{Z}_p$. ConstEq($\boldsymbol{C}_d^{\mathsf{pow}}$, y) homomorphically computes the ciphertext of a Boolean value $(d = y)$ denoted as $C_{(d=y)}$, where

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

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

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

The main algorithm for homomorphic division of a by d in Ref. [21] is based on a simple idea: compute $C_{\lfloor a/y \rfloor} \times C_{(d=y)}$ for all public values $y \in \mathbb{Z}_p$ and take the sum of them:

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

[*8] As mentioned in Ref. [21], this division algorithm can easily be generalized to realize an arbitrary two variable function $f : \mathbb{Z}_p \times \mathbb{Z}_p \to \mathbb{Z}_p$.

Algorithm 4 Div(C_a, C_d)

Require: Ciphertexts: C_a, C_d
Ensure: $C_{\lfloor a/d \rfloor}$

1: $C_{\mathsf{div}} \leftarrow 0$
2: $\boldsymbol{C}_a^{\mathsf{pow}} \leftarrow \mathsf{Pows}(C_a)$
3: $\boldsymbol{C}_d^{\mathsf{pow}} \leftarrow \mathsf{Pows}(C_d)$
4: **for** $i = 0$ to $(p-1)$ **do**
5: $\quad C_{\lfloor a/i \rfloor} \leftarrow \mathsf{ConstDiv}(\boldsymbol{C}_a^{\mathsf{pow}}, i)$
6: $\quad C_{(d=i)} \leftarrow \mathsf{ConstEq}(\boldsymbol{C}_d^{\mathsf{pow}}, i)$
7: $\quad C_{\mathsf{div}} \leftarrow C_{\mathsf{div}} + \mathsf{FHE.Mult}(C_{\lfloor a/i \rfloor}, C_{(d=i)})$
8: **end for**
9: Output C_{div} $(= C_{\lfloor a/d \rfloor})$.

Algorithm 5 Pows(C_a)

Require: Ciphertext: C_a
Ensure: $\boldsymbol{C}_a^{\mathsf{pow}} = (C_a, C_a^2, \ldots, C_a^{p-1})$

1: Let $\ell := \lfloor \log p \rfloor$
2: **for** $i = 0$ to $(\ell - 1)$ **do**
3: $\quad$ **for** $j = 1$ to 2^i **do**
4: $\quad\quad C_a^{2^i+j} \leftarrow \mathsf{FHE.Mult}(C_a^{2^i}, C_a^j)$
5: $\quad$ **end for**
6: **end for**
7: **if** $2^\ell < p - 1$ **then**
8: $\quad$ **for** $i = 1$ to $p - 1 - 2^\ell$ **do**
9: $\quad\quad$ Compute $C_a^{2^\ell+i} \leftarrow \mathsf{FHE.Mult}(C_a^{2^\ell}, C_a^i)$.
10: $\quad$ **end for**
11: **end if**
12: Output $\boldsymbol{C}_a^{\mathsf{pow}} := (C_a, C_a^2, \ldots, C_a^{p-1})$

Main algorithm. Div(C_a, C_d): Algorithm 4 shows the integer-wise homomorphic division algorithm. Div(C_a, C_d) takes two ciphertexts C_a, C_d encrypting plaintexts $a, d \in \mathbb{Z}_p$ as input, and homomorphically computes and outputs the ciphertext $C_{\lfloor a/d \rfloor}$, the decryption of which gives $\lfloor a/d \rfloor$.

The algorithm first invokes Pows for both C_a and C_d to obtain the powers $\boldsymbol{C}_a^{\mathsf{pow}}$, $\boldsymbol{C}_d^{\mathsf{pow}}$. Then, in the for loop, it computes $C_{\lfloor a/i \rfloor}$, $C_{(d=i)}$ by ConstDiv, ConstEq respectively for all $i \in \{0, 1, 2, \ldots, p-1\}$, and performs the homomorphic multiplication for the product $C_{\lfloor a/i \rfloor} \times C_{(d=i)} := \mathsf{FHE.Mult}(C_{\lfloor a/i \rfloor}, C_{(d=i)})$. The product is $C_{\lfloor a/d \rfloor} \times C_1 = C_{\lfloor a/d \rfloor}$ if $i = d$, and otherwise $C_{\lfloor a/i \rfloor} \times C_0 = C_0$. As a result, after the for loop, the output ciphertext C_{div} equals $C_{\lfloor a/d \rfloor}$.

Pows(C_a): Algorithm 5 shows the sub-algorithm Pows(C_a). This algorithm takes a ciphertext C_a encrypting $a \in \mathbb{Z}_p$ as input, and homomorphically computes the powers $\boldsymbol{C}_a^{\mathsf{pow}} := \{C_a, C_a^2, C_a^3, \ldots, C_a^{p-1}\}$. The multiplicative depth of this algorithm is $\log(p-1)$.

ConstDiv($\boldsymbol{C}_a^{\mathsf{pow}}$, y): Algorithm 6 shows the sub-algorithm ConstDiv($\boldsymbol{C}_a^{\mathsf{pow}}$, y). This algorithm takes the 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 $y \in \mathbb{Z}_p$ as input, and homomorphically computes a ciphertext $C_{\lfloor a/y \rfloor}$, the decryption of which gives $\lfloor a/y \rfloor$. We note that Step 1 of Algo-

Algorithm 6 ConstDiv($\boldsymbol{C}_a^{\mathsf{pow}}$, y)

Require: $\boldsymbol{C}_a^{\mathsf{pow}}$: Powers of ciphertext C_a. y: public divisor.

Ensure: $C_{\lfloor a/y \rfloor}$

1: (Precomputing): We define the division function by a constant y as

$$g_y(x) := \left\lfloor \frac{x}{y} \right\rfloor.$$

Given points $\{g_y(x)\}_{x\in\mathbb{Z}_p}$, compute the interpolated polynomial $f_y(x)$ mod p that satisfies $\forall x \in \mathbb{Z}_p, f_y(x) = g_y(x)$ and the coefficient vector $\boldsymbol{a}_{f_y}^{\mathsf{ConstDiv}}(= \{a_{y,0}, a_{y,1}, \ldots, a_{y,p-1}\})$. For all of $y \in \mathbb{Z}_p$, we can compute $\boldsymbol{a}_{f_y}^{\mathsf{ConstDiv}}$ and store $\{\boldsymbol{a}_{f_0}^{\mathsf{ConstDiv}}, \ldots, \boldsymbol{a}_{f_{p-1}}^{\mathsf{ConstDiv}}\}$ as a constant matrix in advance.

2: Compute and output $C_{\lfloor a/y \rfloor} \leftarrow \langle \boldsymbol{a}_{f_y}^{\mathsf{ConstDiv}}, \boldsymbol{C}_a^{\mathsf{pow}} \rangle$ $= \sum_{i=0}^{p-1} \mathsf{FHE.multConst}(a_{y,i}, C_a^i)$

Algorithm 7 ConstEq($\boldsymbol{C}_d^{\mathsf{pow}}, y$)

Require: $\boldsymbol{C}_d^{\mathsf{pow}}$: Powers of ciphertext C_d. y: public constant.

Ensure: $C_{(d=y)}$

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

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

Given points $\{g_y(x)\}_{x\in\mathbb{Z}_p}$, compute the interpolated polynomial $f_y(x)$ that satisfies $\forall x \in \mathbb{Z}_p, f_y(x) = g_y(x)$ and the coefficient vector $\boldsymbol{a}_{f_y}^{\mathsf{ConstEq}}(= \{a_{y,0}, a_{y,1}, \ldots, a_{y,p-1}\})$. For all of $y \in \mathbb{Z}_p$, we compute $\boldsymbol{a}_{f_y}^{\mathsf{ConstEq}}$ and store $\{\boldsymbol{a}_{f_0}^{\mathsf{ConstEq}}, \ldots, \boldsymbol{a}_{f_{p-1}}^{\mathsf{ConstEq}}\}$ as a constant matrix in advance.

2: Compute and output $C_{(d=y)} \leftarrow \langle \boldsymbol{a}_{f_y}^{\mathsf{ConstEq}}, \boldsymbol{C}_d^{\mathsf{pow}} \rangle$ $= \sum_{i=0}^{p-1} \mathsf{FHE.multConst}(a_{y,i}, C_d^i)$

rithm 6 can be precomputed and the constant matrix consisting of polynomial coefficients can be stored before running Div. Further note that this algorithm does not perform homomorphic multiplication (FHE.Mult), and it only requires constant multiplication (FHE.multConst) and homomorphic addition (FHE.Add) whose complexities are smaller than that of FHE.Mult.

ConstEq($\boldsymbol{C}_d^{\mathsf{pow}}, y$): Algorithm 7 shows the sub-algorithm ConstEq($\boldsymbol{C}_d^{\mathsf{pow}}, y$). This algorithm takes the powers of ciphertext $\boldsymbol{C}_d^{\mathsf{pow}} := \{C_d, C_d^2, C_d^3, \ldots, C_d^{p-1}\}$ and a plaintext input $y \in \mathbb{Z}_p$ as input, and homomorphically computes a ciphertext $C_{(d=y)}$, the decryption of which gives 1 if $d = y$ and 0 if $d \neq y$. We note that Step 1 of Algorithm 7 can be precomputed and the constant matrix consisting of polynomial coefficients can be stored before running Div. Further note that this algorithm does not perform homomorphic multiplication (FHE.Mult), and it only requires constant multiplication (FHE.multConst) and homomorphic addition (FHE.Add) whose complexities are smaller than that of FHE.Mult.

4.2 Our Homomorphic Division Algorithm

First we make an observation regarding Table 4, which includes the performance results of homomorphic division. From Table 4, we can see that the ratio of the processing time for polynomial evaluation to the overall processing time is over 80% in the algorithm of Ref. [21], so focusing on reducing the complexity for polynomial evaluation can lead to the significant decrease of the overall processing time. In general, polynomial evaluation consists of homomorphic multiplication (FHE.Mult), homomorphic addition (FHE.Add), constant multiplication (FHE.addConst), and constant addition (FHE.mulConst), and FHE.Mult is the most expensive operation among the basic operations, so reducing the number of invocations of FHE.Mult is a general effective strategy. By following this strategy, in Ref. [21], polynomial $f(x)$ is evaluated with a ciphertext C by computing all the powers $\{C, C^2, \ldots, C^{p-1}\}$ in advance, and these powers are reused to evaluate many different polynomials. Hence once the powers $\{C, C^2, \ldots, C^{p-1}\}$ are computed, the rest of polynomial evaluation can be done without invoking FHE.Mult, and this way of computing can reduce the number of invocations of expensive homomorphic multiplications. However, at the same time, it increases the number of invocations of other inexpensive operations in large. Here we propose an algorithm that replaces ConstEq (Algorithm 7), which is based on polynomial evaluation computing the powers in advance, with FermatEq (Algorithm 9), which is based on Fermat's little theorem. By using our FermatEq, the number of invocations of homomorphic multiplications increases, but the number of invocations of other inexpensive operations can decrease significantly, and as a result, we can decrease the overall processing time (see the implementation result in Section 5.2 for details). Our FermatDiv (Algorithm 8) just replaces ConstEq invoked by Div with FermatEq.

Algorithm 8 FermatDiv(C_a, C_d)

Require: Ciphertexts: C_a, C_d

Ensure: $C_{\lfloor a/d \rfloor}$

1: $C_{\mathsf{div}} \leftarrow 0$
2: $\boldsymbol{C}_a^{\mathsf{pow}} \leftarrow \mathsf{Pows}(C_a)$
3: **for** $i = 0$ to $(p-1)$ **do**
4: $\quad C_{\lfloor a/i \rfloor} \leftarrow \mathsf{ConstDiv}(\boldsymbol{C}_a^{\mathsf{pow}}, i)$
5: $\quad C_{(d=i)} \leftarrow \mathsf{FermatEq}(C_d, i)$
6: $\quad C_{\mathsf{div}} \leftarrow C_{\mathsf{div}} + \mathsf{FHE.Mult}(C_{\lfloor a/i \rfloor}, C_{(d=i)})$
7: **end for**
8: Compute and output C_{div} $(= C_{\lfloor a/d \rfloor})$.

Algorithm 9 FermatEq(C_d, y)

Require: Ciphertexts: C_d. y: public constant.

Ensure: $C_{(d=y)}$

1: $C_{\mathsf{eq}} \leftarrow C_d - y$
2: Compute by homomorphic multiplication and output $C_{(d=y)} \leftarrow 1 - (C_{\mathsf{eq}})^{p-1}$

FermatDiv (Algorithm 8) is constructed based on the following subroutines:

- (Algorithm 5) Pows(C_a): Compute powers of ciphertext C_a.
- (Algorithm 6) ConstDiv($\boldsymbol{C}_a^{\mathsf{pow}}$, y): Integer-wise division of the ciphertext C_a by a public divisor y
- (Algorithm 9) FermatEq(C_d, y): Integer-wise equality check of the ciphertext C_d with a public input y

The main improvement in this algorithm is to replace ConstEq (Algorithm 7) in Ref. [21] with FermatEq (Algorithm 9) using Fermat's little theorem. Therefore, here we omit the detailed explanation of ConstDiv and Pows because we use the same algorithm as in Ref. [21]. For FermatDiv (Algorithm 9), similarly we omit the detailed explanation of the algorithm because ConstEq

is simply replaced by FermatEq.

FermatEq(C_d, y): Algorithm 9 shows the sub-algorithm FermatEq(C_d, y). This algorithm takes a ciphertext C_d and a plaintext input $y \in \mathbb{Z}_p$ as input, and homomorphically computes a ciphertext $C_{(d=y)}$, the decryption of which gives 1 if $d = y$, and 0 if $d \neq y$. In FermatEq, by using Fermat's little theorem, we compute $(C_a - y)^{p-1}$ where

$$(C_d - y)^{p-1} = \begin{cases} C_1 & (\text{if } y \neq d) \\ C_0 & (\text{if } y = d). \end{cases}$$

In our context, FermatEq needs to satisfy the following.

$$\mathsf{FermatEq}(C_d, y) = \begin{cases} C_1 & (\text{if } y = d) \\ C_0 & (\text{if } y \neq d) \end{cases}$$

Hence it follows that FermatEq(C_d, y) can be computed as

$$\mathsf{FermatEq}(C_d, y) = 1 - (C_d - y)^{p-1}.$$

Here we describe the differences between Ref. [21] and our algorithm in more detail. In Div (Algorithm 4) of Ref. [21], the subroutine ConstEq (Algorithm 7) requires $O(p)$ invocations of homomorphic multiplications of the multiplicative depth $\log(p-1)$ to compute the powers of the ciphertext C_d. Further ConstEq requires $O(p)$ invocations of constant multiplications and additions to perform polynomial evaluation. Since ConstEq is invoked p times in a for loop in Div, Div requires $O(p^2)$ invocations of constant multiplications and additions regarding ConstEq. The computation regarding the powers of the ciphertext C_a and the subroutine ConstDiv does not change between Ref. [21] and our algorithm. On the other hand, in our FermatDiv (Algorithm 8), the subroutine FermatEq (Algorithm 9) does not require the computation of the powers of the ciphertext C_d, and instead FermatEq requires $O(\log p)$ invocations of homomorphic multiplications of the multiplicative depth $\log(p-1)$ to compute $(C_d - y)^{p-1}$. Since FermatEq is invoked p times in a for loop in FermatDiv, FermatDiv requires $O(p \log p)$ invocations of homomorphic multiplications regarding FermatEq.

As a result, our FermatDiv needs more invocations of homomorphic multiplications compared with Div of Ref. [21]. However, at the same time, replacing ConstEq with FermatEq enables us to get rid of $O(p^2)$ invocations of constant multiplications and additions, and this riddance can lead to a performance improvement (see Section 5.2 for details).

5. Implementation and Experimental Results

5.1 Integer-wise Homomorphic Comparison

Multiplicative depth. In the previous work by Iliashenko and Zucca [15], the multiplicative depth of Pows (Algorithm 5) to compute the powers of a ciphertext encrypting an ℓ-bit plaintext is $\lceil \log(p-1) \rceil = O(\log p) = O(\ell)$. In polynomial evaluation, the algorithm does not use homomorphic multiplication (FHE.Mult) whose complexity is large. Therefore, the overall multiplicative depth is $O(\ell)$.

In our algorithm, the multiplicative depth to compute the odd powers of a ciphertext encrypting an ℓ-bit plaintext is the same as the multiplicative depth of Pows, which is $\lceil \log(p-1) \rceil = O(\log p) = O(\ell)$.

Experimental environment. We implement our homomorphic comparison using BGV [3] [*9] in HElib and compare the processing time with Ref. [15]. Let the security parameter λ be 120. Let the bit length of the input integer be ℓ. Let the Hamming weight of the secret key ω be 64. The level parameter L should be at least larger than the necessary multiplicative depth. For BGV in HElib, the parameter called "number of bits of the modulus chain" [*10] needs to be specified, so we inlcude this parameter in **Table 2**. The performance evaluation was done on a single-thread PC with a 2 GHz Intel Core i5 and 8 GB RAM.

Result. Table 2 shows the performance evaluation results of Ref. [15] and our algorithm. Introducing an odd polynomial can halve the number of invocations of homomorphic multiplications without changing the multiplicative depth. We can see that our homomorphic comparison without the PS-method is faster than Ref. [15] without the PS-method by a factor of about 5.12 in the plaintext space $\mathbb{Z}_{257}$.

We recall that we perform polynomial evaluations in computing the polynomial of Z (in Eq. (3)), and in our HalfCheck computing C_α, C_β and C_γ. Therefore, we can expect that applying the PS-method to these polynomial evaluations can achieve more improvements, and both Ref. [15] and ours succeeded in improving the performance by using the PS-method. The overall result is that the processing time of our homomorphic comparisons is reduced by a factor of about 14.5 compared with Ref. [15] in the plaintext space $\mathbb{Z}_{257}$.

When we apply the PS-method to homomorphic comparison, the values of k and m in Eq. (2) are set to satisfy $(p-3)/2 \leq k \cdot m$ in Ref. [15] and $p - 1 \leq k \cdot m$ in our algorithm. **Table 3** shows our settings of k and m that achieved the fastest processing speed in the experiments.

5.2 Integer-wise Homomorphic Division

Multiplicative depth. In Div (Algorithm 4) of the previous work [21], the multiplicative depth of Pows (Algorithm 5) to compute the powers of a ciphertext encrypting an ℓ-bit plaintext is $\lceil \log(p-1) \rceil = O(\log p) = O(\ell)$. In Div (Algorithm 4), the homomorphic multiplication (FHE.Mult) is performed once per for loop, so the total multiplicative depth is $\lceil \log(p-1) \rceil + 1 = O(\log p + 1) = O(\ell)$. We note that ConstDiv and ConstEq do not increase the multiplicative depth because they do not use the homomorphic multiplications.

In our FermatDiv (Algorithm 9), the multiplicative depth of Pows to compute the powers of a ciphertext encrypting an ℓ-bit plaintext is $\lceil \log(p-1) \rceil = O(\log p) = O(\ell)$. In FermatEq (Algorithm 9), the multiplicative depth is $\lceil \log(p-1) \rceil = O(\log p) = O(\ell)$ with repeated squaring. With the ciphertexts obtained by Pows and FermatEq, the homomorphic multiplication (FHE.Mult) is performed once per for loop, so the total multiplicative depth is $\lceil \log(p-1) \rceil + 1 = O(\log p + 1) = O(\ell)$. We note that ConstDiv does not increase the multiplicative depth because

[*9] Our homomorphic comparison can also work with BFV [9].

[*10] The larger this parameter is, the slower the performance is, and the larger multiplicative depth we need, the larger this parameter needs to be.

Table 2 Performance comparison of homomorphic comparison.

Method	ℓ	p	L	# bits of modulus chain	Time(s)	PS-method
Iliashenko and Zucca [15]	6	67	7	300	41.79	
	7	131	8	300	108.60	
	7	131	8	400	190.44	X
	8	257	9	300	too much noise	
	8	257	9	400	597.27	
	6	67	7	300	37.99	
	7	131	8	300	too much noise	
	7	131	8	400	175.38	✓
	8	257	9	300	too much noise	
	8	257	9	400	573.41	
Our algorithm	6	67	7	300	26.04	
	7	131	8	300	too mucho noise	
	7	131	8	400	75.89	X
	8	257	9	300	too much noise	
	8	257	9	400	116.58	
	6	67	7	300	13.98	
	7	131	8	300	too much noise	
	7	131	8	400	30.60	✓
	8	257	9	300	too much noise	
	8	257	9	400	39.67	

ℓ: the bit length of the input value. p: plaintext modulus L: level parameter.
Time: processing time. security parameter $\lambda = 120$
※"too much noise" means that noise in the final ciphertext is too much to decrypt correctly.

Table 3 Parameter of PS-method in homomorphic comparison.

	plaintext modulus p	k in Eq. (2)	m in Eq. (2)
Iliashenko and Zucca [15]	67	8	4
	131	8	8
	257	16	8
Our algorithm	67	11	6
	131	13	10
	257	16	16

Table 4 Performance comparison of homomorphic division.

Method	ℓ	p	L	Polynomial Evaluation Time(s)	Total Time(s)
Okada et al. [21]	6	67	8	130.57	168.13
	7	131	9	463.95	525.62
	8	257	10	1778.47	1896.23
	9	521	11	12617.54	13052.23
Our algorithm	6	67	8	65.40	191.47
	7	131	9	227.91	469.09
	8	257	10	862.17	1305.14
	9	521	11	6258.39	8167.41

ℓ: the bit length of the input value. p: plaintext modulus L: level parameter.
Total Time: total processing time. security parameter $\lambda = 120$
※The number of bits of modulus chain is 300 in this table.

it does not use the homomorphic multiplication.

Experimental environment. We implement our homomorphic division using BGV [3] [*11] in HElib and compare the processing time with Ref. [21]. Let the security parameter λ be 120. Let the bit length of the input integer be ℓ. Let the Hamming weight of the secret key ω be 64. The level parameter L should be at least larger than the necessary multiplicative depth. The performance evaluation was done on a single-thread PC with a 2 GHz Intel Core i5 and 8 GB RAM.

Result. **Table 4** shows the performance evaluation results of Ref. [21] and our algorithm. We can see that in Table 4, the processing time of our homomorphic division is a bit slower for the plaintext space $p = 67$, but the processing time was reduced by a factor of at least 1.45 compared with Ref. [21] for the plaintext space $p \geq 257$. Also, by halving the number of polynomial interpolations, we succeeded in reducing the processing time for precomputation and the size of the constant matrix consisting of polynomial coefficients to be stored. It is possible to apply the PS-method to both [21] and our division algorithm. However, as mentioned in Section 2.7, when the PS-method is used for evaluating many different polynomials with the same value for variable x, the total number of multiplications can be larger compared with the case where the PS-method is 'not' used and the powers of x, $\{x, x^2, \ldots, x^{n-1}\}$ are computed in advance. Therefore, here we did not apply the PS-method to the two division algorithms.

Detailed reason for improvement. The main improvement regarding our homomorphic division is to replace ConstEq (Algorithm 7) of Ref. [21] with FermatEq (Algorithm 9), which does not require polynomial evaluation and is computed by Fermat's little theorem. Table 4 shows that for homomorphic division in Ref. [21], the ratio of the processing time for polynomial evaluation to the overall processing time is over 80%. Therefore, reducing the complexity of polynomial evaluation can significantly decrease the overall processing time. In Ref. [21], the powers are computed in advance, and used for polynomial evaluation, and

[*11] Our homomorphic division can also work with BFV [9].

Table 5 Processing time of each operation.

Operation	Time(s) (plaintext modulus $p = 67$)	Time(s) (plaintext modulus $p = 257$)
FHE.addConst	0.0154	0.0161
FHE.Add	0.0003	0.0003
FHE.multConst	0.0116	0.0120
FHE.Mult	0.1875	0.1954

Table 6 Approximate # of invocations and time for each operation in ConstEq & FermatEq with $p = 67$.

Operation	ConstEq when invoked by Div		FermatEq when invoked by FermatDiv	
	Approx. # of invocations	Time(s)	Approx. # of invocations	Time(s)
FHE.addConst	67	1.0318	2×67	2.0636
FHE.Add	$(67-2) \times 67$	1.3065	0	0
FHE.multConst	$(67-1) \times 67$	51.2952	0	0
FHE.Mult	67-2	12.1875	$67 \log (67-1)$	87.0375
Total	-	65.8210	-	89.1011

Table 7 Approximate # of invocations and time for each operation in ConstEq & FermatEq with $p = 257$.

Operation	ConstEq when invoked by Div		FermatEq when invoked by FermatDiv	
	Approx. # of invocations	Time(s)	Approx. # of invocations	Time(s)
FHE.addConst	257	4.1377	2×257	8.2754
FHE.Add	$(257-2) \times 257$	19.6605	0	0
FHE.multConst	$(257-1) \times 257$	789.5040	0	0
FHE.Mult	257-2	49.8270	$257 \log (257-1)$	401.7424
Total	-	863.1292	-	410.0178

this way of computing reduced the number of invocations of homomorphic multiplications, but instead it increased the number of invocations of other operations, resulting in a larger processing time unexpectedly as we explain below.

Table 5 shows that the processing times of homomorphic multiplication (FHE.Mult), homomorphic addition (FHE.Add), constant multiplication (FHE.multConst), and constant addition (FHE.addConst) with security = 120 for $p = 67$ and $p = 257$, respectively. From Table 5, reducing the number of invocations of homomorphic multiplication is generally effective because the processing time of homomorphic multiplication is the largest among the four operations, and Ref. [21] followed this design principle.

Tables 6 and **7** show the approximate number of invocations and processing time for each operation in ConstEq and FermatEq when they are invoked by Div and FermatDiv respectively. First Table 6 shows that when $p = 67$, that is, when p is relatively small, the overall processing time of Ref. [21] is smaller than that of our homomorphic division because the processing time of FHE.ADD and FHE.multConst in ConstEq invoked by Div is smaller than that of FHE.Mult in FermatEq invoked in FermatDiv. Meanwhile, Table 7 shows that when $p = 257$, that is, when p becomes relatively large, the overall processing time of Ref. [21] becomes larger than that of our homomorphic division because the processing time of FHE.ADD and FHE.multConst in ConstEq invoked by Div becomes larger than that of FHE.Mult in FermatEq invoked in FermatDiv, thus achieving better efficiency. Hence we can observe that our homomorphic division is more advantageous when the plaintext space p becomes larger.

6. Conclusion

We proposed the improved algorithms for integer-wise homomorphic comparison and division. For homomorphic comparison, we combined odd univariate polynomial evaluation with the truth table constructed for the secure multiparty comparison protocol of Nishide and Ohta [20] compared with Iliashenko and Zucca [15]. As a result, we succeeded in reducing the processing time without the limited input range. In addition, the processing time decreased further by applying the PS-method [24], which sped up the polynomial evaluation. As a result, we succeeded in reducing the processing time of homomorphic comparison by a factor of about 14.5 compared with Ref. [15] when the plaintext modulus p is 257.

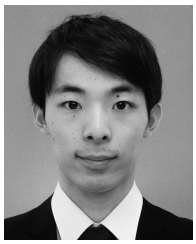
For homomorphic division, we found that most of the processing time in Okada et al. [21] is due to the polynomial evaluations, and we proposed the algorithm to reduce polynomial evaluations. By using Fermat's little theorem, we succeeded in halving the number of polynomial evaluations. Although the number of invocations of homomorphic multiplications required by our homomorphic division increased, the number of invocations of other homomorphic additions and constant multiplications decreased significantly. As a result, we succeeded in reducing the processing time of homomorphic division by a factor of about 1.45 without increasing the multiplicative depth when the plaintext modulus p is 257.

Acknowledgments We would like to thank the anonymous reviewers for their helpful comments to improve this paper. This work was supported in part by JSPS KAKENHI Grant Number 20K11807.

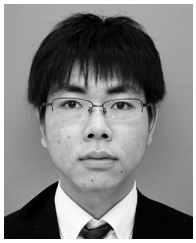
References

[1] Bost, R., Popa, R.A., Tu, S. and Goldwasser, S.: Machine learning classification over encrypted data, *NDSS*, p.14 (2015).
[2] Bourse, F., Minelli, M., Minihold, M. and Paillier, P.: Fast homomorphic evaluation of deep discretized neural networks, *CRYPTO*, pp.483–512, Springer (2018).
[3] Brakerski, Z., Gentry, C. and Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping, *ITCS*, pp.309–325, ACM (2012).
[4] Çetin, G.S., Doröz, Y., Sunar, B. and Savaş, E.: Depth optimized efficient homomorphic sorting, *Latincrypt*, pp.61–80, Springer (2015).
[5] Cheon, J.H., Kim, A., Kim, M. and Song, Y.: Homomorphic encryp-

tion for arithmetic of approximate numbers, *Asiacrypt*, pp.409–437, Springer (2017).
[6] Chillotti, I., Gama, N., Georgieva, M. and Izabachène, M.: TFHE: Fast fully homomorphic encryption over the torus, *Journal of Cryptology*, Vol.33, No.1, pp.34–91 (2020).
[7] Ducas, L. and Micciancio, D.: FHEW: Bootstrapping homomorphic encryption in less than a second, *Eurocrypt*, pp.617–640, Springer (2015).
[8] ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Trans. Information Theory*, Vol.31, No.4, pp.469–472 (1985).
[9] Fan, J. and Vercauteren, F.: Somewhat practical fully homomorphic encryption, Cryptology ePrint Archive, Report 2012/144 (2012).
[10] Gentry, C.: *A fully homomorphic encryption scheme*, PhD thesis, Stanford University (2009).
[11] Gentry, C.: Fully homomorphic encryption using ideal lattices, *STOC*, pp.169–178, ACM (2009).
[12] Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M. and Wernsing, J.: Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy, *ICML*, pp.201–210, PMLR (2016).
[13] Halevi, S. and Shoup, V.: Design and implementation of a homomorphic-encryption library, *IBM Research* (*Manuscript*), Vol.6, pp.12–15 (2013).
[14] Iliashenko, I., Negre, C. and Zucca, V.: Integer functions suitable for homomorphic encryption over finite fields, *Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (*WAHC*), pp.1–10, ACM (2021).
[15] Iliashenko, I. and Zucca, V.: Faster homomorphic comparison operations for BGV and BFV, *PoPETs*, Vol.2021, No.3, pp.246–264 (2021).
[16] Juvekar, C., Vaikuntanathan, V. and Chandrakasan, A.: GAZELLE: A low latency framework for secure neural network inference, *USENIX Security Symposium*, pp.1651–1669 (2018).
[17] Lyubashevsky, V., Peikert, C. and Regev, O.: On ideal lattices and learning with errors over rings, *Eurocrypt*, pp.1–23, Springer (2010).
[18] Morimura, K., Maeda, D. and Nishide, T.: Improved integer-wise homomorphic comparison and division based on polynomial evaluation, *International Conference on Availability, Reliability and Security* (*ARES*), pp.1–10, ACM (2022).
[19] Narumanchi, H., Goyal, D., Emmadi, N. and Gauravaram, P.: Performance analysis of sorting of FHE data: integer-wise comparison vs bit-wise comparison, *AINA*, pp.902–908, IEEE (2017).
[20] Nishide, T. and Ohta, K.: Multiparty computation for interval, equality, and comparison without bit-decomposition protocol, *PKC*, pp.343–360, Springer (2007).
[21] Okada, H., Cid, C., Hidano, S. and Kiyomoto, S.: Linear depth integer-wise homomorphic division, *IFIP International Conference on Information Security Theory and Practice*, pp.91–106, Springer (2018).
[22] Okada, H., Kiyomoto, S. and Cid, C.: Integerwise functional bootstrapping on TFHE, *ISC*, pp.107–125, Springer (2020).
[23] Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes, *Eurocrypt*, pp.223–238, Springer (1999).
[24] Paterson, M.S. and Stockmeyer, L.J.: On the number of nonscalar multiplications necessary to evaluate polynomials, *SIAM Journal on Computing*, Vol.2, No.1, pp.60–66 (1973).
[25] Regev, O.: On lattices, learning with errors, random linear codes, and cryptography, *Journal of the ACM* (*JACM*), Vol.56, No.6, pp.1–40 (2009).
[26] Rivest, R.L., Shamir, A. and Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems, *Comm. ACM*, Vol.26, No.1, pp.96–99 (1983).
[27] Smart, N.P. and Vercauteren, F.: Fully homomorphic SIMD operations, *Designs, Codes and Cryptography*, Vol.71, No.1, pp.57–81 (2014).
[28] Shoup, V.: A library for doing number theory (2016), available from ⟨http://shoup.net/ntl/⟩.



Koki Morimura received B.S. degree from University of Tsukuba in 2021, M.S. degree from the University of Tsukuba in 2023. His research interests are in cryptography and information security.



Daisuke Maeda received B.S. degree from University of Tsukuba in 2021, M.S. degree from the University of Tsukuba in 2023. His research interests are in cryptography and information security.

Takashi Nishide received B.S. degree from the University of Tokyo in 1997, M.S. degree from the University of Southern California in 2003, and Dr.E. degree from the University of Electro-Communications in 2008. From 1997 to 2009, he had worked at Hitachi Software Engineering Co., Ltd. developing security products. From 2009 to 2013, he had been an assistant professor at Kyushu University and from 2013 he is an associate professor at University of Tsukuba. His research interests are in cryptography and information security.