IEICE TRANS. FUNDAMENTALS, VOL.E105–A, NO.3 MARCH 2022

PAPER Special Section on Information Theory and Its Applications

An Efficient Secure Division Protocol Using Approximate Multi-Bit Product and New Constant-Round Building Blocks*

Keitaro HIWATASHI[†,††a)], **Satsuya OHATA**[†††], ***Nonmembers*, *and* Koji NUIDA**[††,††††], ***Member***

SUMMARY Integer division is one of the most fundamental arithmetic operators and is ubiquitously used. However, the existing division protocols in secure multi-party computation (MPC) are inefficient and very complex, and this has been a barrier to applications of MPC such as secure machine learning. We already have some secure division protocols working in $\mathbb{Z}_{2^n}$. However, these existing results have drawbacks that those protocols needed many communication rounds and needed to use bigger integers than in/output. In this paper, we improve a secure division protocol in two ways. First, we construct a new protocol using only *the same size integers* as in/output. Second, we build efficient *constant-round building blocks* used as subprotocols in the division protocol. With these two improvements, communication rounds of our division protocol are reduced to about 36% (87 rounds → 31 rounds) for 64-bit integers in comparison with the most efficient previous one.

key words: *secure multi-party computation, division protocol, client-aided model, constant-round protocols*

1. Introduction

Secure multi-party computation (MPC) is a technique that enables a set of parties to compute a function jointly without revealing their own inputs to the others. MPC has been actively studied since Yao [2] first advocated it. There are several ways to realize MPC; homomorphic encryption (HE), garbled circuit (GC), fully homomorphic encryption (FHE), and secret sharing (SS). Among them, some recent research (e.g., [3], [4]) showed that SS-based MPC could achieve high-throughput and information-theoretic security. Moreover, there are also some publicly accessible implementations of SS-based MPC such as ABY [5]** and SCALE-MAMBA***; such libraries suggest that a real-life use of SS-based MPC would now be within a practical scope. According to these advantages and recent research trends, in this paper, we focus on SS-based MPC.

There are some models in SS-based MPC, and we focus on client-server MPC in this paper. In this model, arbitrary number of clients split their data into shares and send them to $N(\geq 2)$ computation parties (CPs). Then, CPs compute a function jointly and return outputs to the clients. Recent research results on high-speed MPC (e.g., [3], [4]) have mainly treated three-party computation. However, we focus on two-party computation in this paper since fewer hardware resources are better in practice.

There are mainly two types of network environments; local-area network (LAN) and wide-area network (WAN). In the LAN setting, since the latency is very small and the bandwidth is very high, the local computation time affects the total execution time. On the other hand, in the WAN setting, the time for not computation but communication (i.e., latency and data transfer) often occupy most of the total execution time. The computation cost of SS-based MPC is lower than GC-based or (F)HE-based ones since it does not use any heavy (public key) cryptographic tools in some models (see the end of this section for more details). On the other hand, the total latency of GC-based MPC is much smaller than the SS-based one since it requires fewer communication rounds. When we only execute secure division protocol in WAN environments, not SS-based MPC but GC-based one is suitable in most cases. However, when we securely compute some functions in practice, we usually use not only division but standard arithmetic operations (e.g., addition, multiplication). In these situations, the only usage of GC-based MPC takes longer execution time than SS-based one since it is hard to efficiently compute arithmetic operations such as addition or multiplication using (standard) GC-based MPC. Moreover, SS-based MPC can achieve information-theoretic security (as long as the correlated randomness is ideally generated), which is not achievable by GC-based approach. Therefore, we consider it is interesting to propose the tailored construction of the SS-based secure division protocol. To take advantage of SS-based and GC-based MPC, protocol mixing has been proposed (e.g., [5]), and this is undoubtedly a promising approach. However, conversions are not free. Moreover, deriving the optimal mixing is hard in general [6], [7]. For the above reasons, in this paper, we tackle the problem of how we securely and efficiently compute the arithmetic division protocol only using SS.

In this paper, we treat a secure division protocol, which is an important process for many applications. In the (non-

Manuscript received February 14, 2021.
Manuscript revised June 18, 2021.
Manuscript publicized September 28, 2021.

[†]The author is with The University of Tokyo, Tokyo, 113-8654 Japan.
[††]The authors are with the National Institute of Advanced Industrial Science and Technology, Tokyo, 135-0064 Japan.
[†††]The author is with Digital Garage, Inc., Tokyo, 150-0042 Japan.
[††††]The author is with Institute of Mathematics for Industry (IMI), Kyushu University, Fukuoka-shi, 819-0395 Japan.

*A part of this paper was presented at ACNS 2020 [1]. This paper presents more detailed discussion about the correctness of proposed protocols.

a) E-mail: keitaro_hiwatashi@mist.i.u-tokyo.ac.jp

DOI: 10.1587/transfun.2021TAP0004

**https://github.com/encryptogroup/ABY
***https://homes.esat.kuleuven.be/~nsmart/SCALE/

Copyright © 2022 The Institute of Electronics, Information and Communication Engineers

privacy-preserving) training of machine learning models, for example, (1) we usually normalize the data distribution to realize the fast and stable training; and (2) we compute softmax functions (in neural networks) to calculate the loss of the training iteration. In both cases, we cannot avoid calculating division. In other applications such as k-means clustering or chi-squared test, we also need to compute division. Hence, we need to execute a secure division protocol when we construct privacy-preserving machine learning or other above applications. However, secure division protocols are known to be much more massive than other fundamental secure protocols like addition, multiplication, etc. Most of the previous research results on privacy-preserving neural networks treat not training but inferences. This is (probably) because we need an extremely high cost for privacy-preserving training. We cannot doubt that one of the critical reasons for this is the inefficiency of secure arithmetic division protocols. Although there are some previous research results on secure division protocols [8]–[14], all of them are not efficient enough in practice. For example, in [12], we need 87 communication rounds to execute the secure division protocol for 64-bit integers. Moreover, we also need to expand the size of integers to 206-bit during the computation for controlling calculation errors correctly. If we can improve the efficiency of secure division protocols, we can construct privacy-preserving applications more and make them more efficient.

1.1 Our Contribution

We propose an efficient division protocol via the following two approaches.

1. We propose a new construction strategy for secure division protocols. In this strategy, we need not bit size expansion in the protocol; that is, we always treat n-bit integers in our protocol, where n is the bit length of input/output values. This is a remarkable advantage in the ease of implementation (i.e., we do not need to introduce large arithmetic numbers) as well as the practice efficiency. In fact, [15] mentioned that modular addition/multiplication become 100 times slower when we use the libraries for arbitrary-length integers (e.g., GMP, NTL). We can avoid using these libraries and keep computation fast.
2. We construct new constant-round building blocks for secure division protocols. Existing constant-round SS-based protocols (e.g., [16]–[18]) work over $\mathbb{F}_p$. Our proposed arithmetic overflow detection protocol Overflow is the first constant-round protocol working over $\mathbb{Z}_{2^n}$, which is a more natural encoding of finite-precision integers. We can execute our Overflow with constant (in fact, only three) communication rounds.

With these two approaches, we can obtain an efficient secure division protocol. Our protocol only requires 31 communication rounds for 64-bit integers. This is about 64% smaller ($87 \rightarrow 31$) than the previous result [12]. We show the theoretical and experimental evaluation of our protocol in Sect. 5.

The technical overview of these results are as follows:

1.1.1 Secure Division Protocol without Bit Expansion

In the same way as the previous results [8]–[10], [12], we also start from the approach by Goldschmidt [19]. To compute the integer division $\lfloor N/D \rfloor$, the numerator N and the denominator D are iteratively multiplied by common factors in a way that the denominator converges to 1 so that the product at the numerator can be used as an approximated result. To implement this method, the strategy of the previous result [12] is to make the approximation as good as possible and finally add an explicitly estimated correction term to obtain the exact result. However, the requirement of highly accurate approximation caused the following two inefficiency problems; the number of iterated products has to be large, and; for a better approximation of products of n-bit values, intermediate values with not only n-bit but $2n$-bit or even higher accuracy have to be handled (e.g., 206-bit values were needed for 64-bit inputs). To overcome these issues, the key idea of this paper is the following; even if the approximation error is not tiny and cannot be explicitly estimated, once the correct result is guaranteed to be within a reasonably small range, the correct result will be found by a kind of (securely implemented) exhaustive search over this range. Due to the unnecessity of highly accurate approximation, now the number of iterations is decreased, and a product of n-bit values may be computed in a less accurate but more efficient way using only n-bit values; we construct a protocol for the approximate multiplication. Moreover, the protocol is also extended to the multiplication of $M > 2$ values. Here, as M increases, the number of iterations is reduced further, while it becomes more difficult to estimate the range of the error. We determine a value of M with a better trade-off and perform the (non-trivial) error estimation, then obtain a more efficient division protocol. See Sect. 3 for more details.

1.1.2 Constant-Round Building Blocks

We construct a constant-round secure overflow detection protocol Overflow, which is frequently used in the secure division protocol. We consider $x \in \mathbb{Z}_{2^n}$ and its shares $[\![x]\!]_1, [\![x]\!]_2$. Overflow detects whether $[\![x]\!]_1 + [\![x]\!]_2 \geq 2^n$ or not. In the previous results [9], [20], we need $\Theta(\log n)$ communication rounds for executing Overflow since we have to expand the arithmetic share to the binary and check the carry from a right (= smaller) side. When we come to consider the functionality of Overflow, however, it is enough to consider whether the following conditions hold or not. First, we find the leftmost carry position C. Second, we check the condition whether the carry in C propagates to the left edge. In this strategy, we do not need to calculate carries for all bits from the right side. We construct some subprotocols for executing this strategy in practice. Our Overflow only need three communication rounds. For more details, see Sect. 4. Note that, although we can construct two-round

Overflow for 64-bit integers [20], the outputs of this protocol are not arithmetic but bit-wise shares. In many cases, we convert them to arithmetic ones for the next procedures with an additional communication round. We do not need this additional communication since our protocol directly outputs arithmetic shares. Note also that, the round-efficient Overflow in [20] is based on several multi-fan-in AND/OR gates, which results in larger computation and memory costs.

1.1.3 A Note on Client-Aided Model

In this paper, we adopt the client-aided model [15], [17], [20] for client-server SS-based MPC, which is a kind of trusted dealer setup model. More precisely, in this model, the clients still do not participate in the online computation phase of the protocol, while in the pre-computation phase, the clients send to the servers not only their shared inputs but also certain kinds of auxiliary information (i.e., Beaver triples) we use in the protocol. Although the clients will have to perform some more computations and communications, this model has an advantage that any complicated auxiliary information required in some advanced protocols can be easily provided (in comparison to the simple two-server case where the servers themselves have to generate it by using some additional cryptographic machinery), which yields significant decreases of the communication rounds. We note that the performance comparison (e.g., for numbers of communication rounds) in this paper is based on this model.

2. Preliminaries and Settings

In this section, we review basic notations and techniques on which our secure division protocol is based.

2.1 Notations

$x \overset{\mathrm{R}}{\in} A$ means x is chosen from set A uniformly at random. In this paper, we mainly treat n bit integers. $x[i]$ for n-bit integer x is the i-th least significant bit of x. That is, $x = \sum_{i=1}^{n} x[i]2^{i-1}$. Also, $x[t...1]$ for n-bit integer x means $x \bmod 2^t$. We use bold letter to express an array. For array $\mathbf{X}$, $\mathbf{X}[i]$ is the i-th element of $\mathbf{X}$. We treat boolean values True as 1 and False as 0, respectively.

2.2 Secret Sharing

A 2-out-of-2 secret sharing over $\mathbb{Z}_{2^n}$ consists of two algorithms called Share and Reconst. Share has an input $x \in \mathbb{Z}_{2^n}$ and computes $(\llbracket x \rrbracket_1, \llbracket x \rrbracket_2)$, where $\llbracket x \rrbracket_1 \overset{\mathrm{R}}{\in} \mathbb{Z}_{2^n}$ and $\llbracket x \rrbracket_2 = x - \llbracket x \rrbracket_1 \bmod 2^n$. Reconst has an input $(\llbracket x \rrbracket_1, \llbracket x \rrbracket_2)$ and computes $x = \llbracket x \rrbracket_1 + \llbracket x \rrbracket_2 \bmod 2^n$. $\llbracket x \rrbracket_i$ is the share of i-th party.

Using this secret sharing scheme, we can realize affine operations without any communications[†], and multiplication with auxiliary inputs called Beaver triplet. Beaver triplet is a set of shares $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$ such that a and b are random values not known by each party and c equals ab. We use $\llbracket \cdot \rrbracket \times \llbracket \cdot \rrbracket$ for representing this secure multiplication.

[†]Linear operations are realized by computing the linear operations locally, and adding some constant a is realized by adding a share $(a, 0)$.

2.3 Adversary Model

In this paper, we assume semi-honest adversaries. That is, even a corrupted party follows protocols precisely. The simulation-based security notion in the presence of semi-honest adversaries is defined as Definition 1 [21].

Definition 1. *Let $f : (\{0,1\}^*)^2 \to (\{0,1\}^*)^2$ be a probabilistic 2-ary functionality and $f_i(\vec{x})$ denotes the i-th element of $f(\vec{x})$ for $\vec{x} = (x_0, x_1) \in (\{0,1\}^*)^2$ and $i \in \{0,1\}$; $f(\vec{x}) = (f_0(\vec{x}), f_1(\vec{x}))$. Let Π be a 2-party protocol to compute the functionality f. The view of party P_i for $i \in \{0,1\}$ during an execution of Π on input $\vec{x} = (x_0, x_1) \in (\{0,1\}^*)$ where $|x_0| = |x_1|$, denoted by $\mathrm{VIEW}_i^{\Pi}(\vec{x})$, consists of $(x_i, r_i, m_{i,1}, \ldots, m_{i,t})$, where x_i represents P_i's input, r_i represents its internal random coins, and $m_{i,j}$ represents the j-th message that P_i has received. The output of all parties after the execution of Π on input $\vec{x}$ is denoted as $\mathrm{OUTPUT}^{\Pi}(\vec{x})$. Then for each party P_i, we say that* Π privately computes *f in the presence of semi-honest corrupted party P_i if there exists a probabilistic polynomial-time algorithm $\mathcal{S}$ such that*

$$\{(\mathcal{S}(i, x_i, f_i(\vec{x})), f(\vec{x}))\} \equiv \{(\mathrm{VIEW}_i^{\Pi}(\vec{x}), \mathrm{OUTPUT}^{\Pi}(\vec{x}))\}$$

where the symbol $\equiv$ means that the two probability distributions are statistically indistinguishable.

Affine operations and multiplication treated in Sect. 2.2 are known to be semi-honest secure. Also, as described in [21], Composition Theorem for the semi-honest model holds; that is, any protocol is privately computed as long as the protocol is privately computed assuming the ideal subroutines and its actual subroutines are privately computed. For this reason, we do not discuss the security of protocols in the rest of this paper.

2.4 Building Blocks

Here, we introduce functionalities of protocols and summarize their communication rounds in [22]. MSNZB is an acronym of Most Significant Non Zero Bit. See [22] for more details. Note that in [22], shares of a boolean value were bit-wise shares (that is, $x = \llbracket x \rrbracket_1 \oplus \llbracket x \rrbracket_2$, where $\oplus$ means exclusive OR), while these are arithmetic shares in this paper as mentioned Sect. 2.2.

- Overflow : $\llbracket y \rrbracket \leftarrow \mathrm{Overflow}(\llbracket x \rrbracket, i)$, where y is the boolean value corresponding to $(\llbracket x \rrbracket_1 \bmod 2^i) + (\llbracket x \rrbracket_2 \bmod 2^i) \overset{?}{\geq} 2^i$. It takes $1 + \lceil \log_2 n \rceil$ rounds.
- ExtractBit : $\llbracket y \rrbracket \leftarrow \mathrm{ExtractBit}(\llbracket x \rrbracket, i)$, where y is equal to $x[i]$. It takes $1 + \lceil \log_2 n \rceil$ rounds.
- RightShift : $\llbracket y \rrbracket \leftarrow \mathrm{RightShift}(\llbracket x \rrbracket, i)$, where y is the i-bit right shift of x. It takes $2 + \lceil \log_2 n \rceil$ rounds.

- Comparison : $[\![z]\!] \leftarrow \mathsf{Comparison}([\![x]\!], [\![y]\!])$, where z is the boolean value corresponding to $x \overset{?}{<} y$. It takes $3 + \lceil \log_2 n \rceil$ rounds.
- Equal_zero : $[\![y]\!] \leftarrow \mathsf{Equal_zero}([\![x]\!])$, where y is the boolean value corresponding to $x \overset{?}{=} 0$. It takes $\lceil \log_2 n \rceil$ rounds.
- MSNZB : $([\![y_1]\!], \ldots, [\![y_m]\!]) \leftarrow \mathsf{MSNZB}([\![x_1]\!], \ldots, [\![x_m]\!])$ (each x_i is equal to 0 or 1), where $([\![y_1]\!], \ldots, [\![y_m]\!])$ satisfies the equations below:

$$y_i = \begin{cases} 1 & x_i = 1,\ x_j = 0\ (\forall j < i) \\ 0 & \text{otherwise.} \end{cases}$$

It takes $\lceil \log_2 n \rceil$ rounds.

3. Construction of Division Protocol

In this section, we construct a new division protocol. Secure division takes two shares $[\![N]\!], [\![D]\!]$ as inputs and returns a share of the quotient $\lfloor N/D \rfloor$. Here, we assume $D \neq 0$.

3.1 Goldschmidt's Method

Existing methods [8]–[10], [12] are based on Goldschmidt's division algorithm [21]. Goldschmidt's division algorithm computes a quotient by multiplying iteratively both the numerator and denominator by the same factors Y_i,

$$\frac{N}{D} = \frac{NY_0Y_1 \cdots}{DY_0Y_1 \cdots},$$

so that the denominator converges to 1. In many cases, Y_i is chosen as below:

1. $Y_0 = 2^{-d}$, where d is the bit length of D.
2. $\varepsilon = 1 - Y_0 D$, $Y_i = 1 + \varepsilon^{2^{i-1}}$ $(i \geq 1)$.

In this paper, we take Y_i $(i \leq \lceil \log_2 n \rceil)$ into consideration, and approximate the quotient N/D by

$$\frac{N}{D} \approx N2^{-d}(1 + \varepsilon + \cdots + \varepsilon^n).$$

Technically, $Y_1Y_2 \cdots Y_{\lceil \log_2 n \rceil}$ is equal to $1+\varepsilon+\cdots+\varepsilon^{2^{\lceil \log_2 n \rceil}-1}$, but we ignore terms after ε^n (note that $0 < \varepsilon \leq \frac{1}{2}$). We need to deal with decimals in this method, and we express decimals by rounded integers obtained by multiplying the decimals by $2^{n'}$ for a certain parameter n'. We express the result integer by $\hat{\cdot}$. For example, $\hat{x} = 3$ for $x = 0.375$ and $n' = 3$.

[12] constructed a two-party protocol computing Goldschmidt's method (Protocol 1[†]).

Here, ReciprocalGuess is a protocol which computes $2^{n'-d}$ with the same number of communication rounds as Comparison, where d is the bit length of the input D.

[†]The symbol $\widehat{\cdot}$ in step 7, step 8 and step 12 means the decimal is multiplied by $2^{2n'}$, instead of $2^{n'}$

Protocol 1 Divide [12]

Input: $[\![N]\!], [\![D]\!]$, and parameters $h_0 = \lceil \log_2(n+2) \rceil - 1$, $n' = n + 2 + \lceil \log_2(3h_0) \rceil$, $m = n + 2n'$
Output: $[\![Q]\!]$, where $Q = \lfloor \frac{N}{D} \rfloor$

1: $[\![\widehat{Y_0}]\!] \leftarrow \mathsf{ReciprocalGuess}([\![D]\!], n')$
2: $[\![N]\!] \leftarrow \mathsf{CastUp}_{2^n \to 2^m}([\![N]\!])$, $[\![D]\!] \leftarrow \mathsf{CastUp}_{2^n \to 2^m}([\![D]\!])$
3: $[\![\widehat{N_0}]\!] \leftarrow [\![N]\!] \times [\![\widehat{Y_0}]\!]$, $[\![\widehat{D_0}]\!] \leftarrow [\![D]\!] \times [\![\widehat{Y_0}]\!]$
4: $[\![\widehat{\varepsilon}]\!] \leftarrow \widehat{1} - [\![\widehat{D_0}]\!]$, where $\widehat{1} = 2^{n'} \cdot 1$
5: $[\![\widehat{Y_1}]\!] \leftarrow \widehat{1} + \widehat{\varepsilon}$
6: **for** $h = 1, \ldots, h_0$ **do**
7: $\quad [\![\widehat{\widehat{N_h}}]\!] \leftarrow [\![\widehat{N_{h-1}}]\!] \times [\![\widehat{Y_h}]\!]$, $[\![\widehat{\widehat{\varepsilon^{2^h}}}]\!] \leftarrow [\![\widehat{\varepsilon^{2^{h-1}}}]\!] \times [\![\widehat{\varepsilon^{2^{h-1}}}]\!]$
8: $\quad [\![\widehat{N_h}]\!] \leftarrow \mathsf{RightShift}([\![\widehat{\widehat{N_h}}]\!], n')$, $[\![\widehat{\varepsilon^{2^h}}]\!] \leftarrow \mathsf{RightShift}([\![\widehat{\widehat{\varepsilon^{2^h}}}]\!], n')$
9: $\quad [\![\widehat{Y_{h+1}}]\!] \leftarrow \widehat{1} + [\![\widehat{\varepsilon^{2^h}}]\!]$
10: **end for**
11: $[\![\Delta]\!] \leftarrow 2^{n'-n}[\![\widehat{Y_0}]\!] \times [\![N]\!]$, $[\![\widehat{\widehat{N_{h_0+1}}}]\!] \leftarrow [\![\widehat{N_{h_0}}]\!] \times [\![\widehat{Y_{h_0+1}}]\!]$
12: $[\![Q]\!] \leftarrow \mathsf{RightShift}([\![\widehat{\widehat{N_{h_0+1}}}]\!] + [\![\Delta]\!], 2n')$

$\mathsf{CastUp}_{2^n \to 2^m}$ is a protocol which converts a share over $\mathbb{Z}_{2^n}$ to a share over $\mathbb{Z}_{2^m}$ with the same number of communication rounds as Overflow. See [12] for more details.

[12] used $n' = n + 2 + \lceil \log_2 3(\lceil \log_2(n+2) \rceil - 1) \rceil$ as the parameter for expressing decimals, and needed to deal with larger integers whose bit length is equal to $n+2n'$. From now, we construct a two-party protocol computing Goldschmidt's method without bit expansion mentioned above. We let n' be equal to n[††].

3.2 Approximate Multi-Bit Product – MultBit Protocol

We construct MultBit protocol which computes a product of decimals approximately. In [12], bit expansion was needed in calculating the product of decimals. The notable point is that RightShift was applied after the product of decimals. This is because the product of two decimals is multiplied by $2^{2n'}$, instead of $2^{n'}$. Taking into consideration the fact that RightShift is applied after product, we can construct an approximate protocol without bit expansion (Protocol 2). The idea is, in the equations below, we replace $x2^{-i}$ with $\mathsf{RightShift}(x)$:

$$\begin{aligned} xy2^{-n} &= x2^{-n} \sum_{i=1}^{n} y[n-i+1]2^{n-i} \\ &= \sum_{i=1}^{n} x2^{-i} y[n-i+1]. \end{aligned}$$

However, rounding error in this way becomes bigger than in computing with bit expansion. This rounding error is estimated in Sect. 3.5 in detail.

3.3 Multi-Fan-in MultBit Protocol

Though the protocol above has two inputs, we can extend

[††]As a natural consequence of not expanding bit size, n' should be at most n. Hence, we let n' be equal to n so that a rounding error is minimal.

Protocol 2 MultBit

Input: $[\![x]\!], [\![y]\!]$
Output: $[\![z]\!]$, where $z \approx xy2^{-n}$
1: **for** $i \in \{1, 2, \ldots, n-1\}$ **do**
2: $\quad [\![x_i]\!] \leftarrow \mathsf{RightShift}([\![x]\!], i)$
3: **end for**
4: $[\![z]\!] \leftarrow \sum_{i=1}^{n} [\![x_i]\!] \mathsf{ExtractBit}([\![y]\!], n-i+1)$

Protocol 3 M_MultBit

Input: $[\![x]\!], [\![y_1]\!], \ldots, [\![y_{M-1}]\!]$
Output: $[\![z]\!]$, where $z \approx x \prod_{i=1}^{M-1}(y_i 2^{-n})$
1: **for** $i \in \{1, 2, \ldots, n-1\}$ **do**
2: $\quad [\![x_i]\!] \leftarrow \mathsf{RightShift}([\![x]\!], i)$
3: **end for**
4: $[\![z]\!] \leftarrow \sum_{i=1}^{n} \sum_{i_1+\cdots+i_{M-1}=i} [\![x_i]\!] \prod_{j=1}^{M-1} \mathsf{ExtractBit}([\![y_j]\!], n-i_j+1)$

it for multiple (more than two) inputs. That is, we use the deformation below:

$$xyz2^{-2n} = \sum_{(i,j)\in\{1,\ldots,n\}^2} x2^{-i-j} y[n-i+1]z[n-j+1].$$

Although this is the case of three inputs, we can do in the case of M inputs in the same way. Using such multi-fan-in products, the number of iterations in Goldschmidt's method, hence the total communication rounds, are reduced. However, the computation cost and the communication size grow exponentially with respect to M. In this paper, we use M-fan-in MultBit (M_MultBit) for $M \leq 4$. The detail of M_MultBit is given in Protocol 3. Also, the term $\sum_{i_1+\cdots+i_{M-1}=i} \prod_{j=1}^{M-1} \mathsf{ExtractBit}([\![y_j]\!], n-i_j+1)$ can be regarded as a convolution. Therefore, we can compute this term very efficiently using Number Theoretic Transform (NTT), which is a kind of discrete Fourier transform. Note that since NTT is a linear transformation, we can locally compute it. (NTT is used in other privacy-preserving protocols (e.g., [23]).) One may think that we cannot compute NTT using shares over $\mathbb{Z}_{2^n}$ naively since it is a linear transformation over $\mathbb{F}_p$. However, since our new ExtractBit can output the shares over $\mathbb{F}_p$ (see Sect. 4 for more details), we can use NTT. Since we use M_MultBit in the case of $y_1 = y_2 = \cdots = y_{M-1}$ in this paper, we express M_MultBit$([\![x]\!], [\![y]\!], \ldots, [\![y]\!])$ by M_MultBit$([\![x]\!], [\![y]\!])$ for short.

3.4 Goldschmidt's Method Using Multi-Fan-in MultBit

Here, we construct Goldschmidt's method with M_MultBit. First, we construct a protocol Power as in Protocol 4 which approximately computes the m-th power of the input. Recall that the output of $(j+1)$_MultBit$([\![x]\!], [\![y]\!])$ is a share of an approximate value of $xy^j \cdot 2^{-(jn)}$. Hence, $\widehat{\delta}_{4^{i-1}j+k}$ in step 6 of Protocol 4 is an approximate value of $\widehat{\delta_k}\widehat{\delta}^j_{4^{i-1}} \cdot 2^{-jn}$ and $\widehat{\delta_i}$ is an approximate value of $\widehat{\varepsilon}^i 2^{-(i-1)n}$ for $i = 1, \ldots, m$ inductively. Second, using Power, we construct a protocol QGuess as in Protocol 5 which approximately computes Goldschmidt's method.

Roughly speaking, QGuess protocol corresponds to the for-loop in Protocol 1. The error in the for-loop in Protocol 1 is small enough to the error can be calculated (in step 11 in Protocol 1). On the other hand, the error in our QGuess protocol is a little bit large. However, we give the upper bound of the error (instead of the formula of the exact error) and we can correct the error by the method described in Sect. 3.6. Also, our QGuess is more round-efficient than the for-loop in Protocol 1 in return for the large error.

Protocol 4 Power

Input: $[\![\widehat{\varepsilon}]\!], m$
Output: $([\![\widehat{\delta}_1]\!], \ldots, [\![\widehat{\delta}_m]\!])$, where $\widehat{\delta_i} \approx (\widehat{\varepsilon})^i 2^{-(i-1)n}$ $(i = 1, \ldots, m)$
1: $[\![\widehat{\delta}_1]\!] \leftarrow [\![\widehat{\varepsilon}]\!]$
2: **for** $i = 1, 2, \ldots, \lceil \log_4 m \rceil$ **do**
3: $\quad$ **for** $j = 1, 2, 3$ **do**
4: $\quad\quad$ **for** $k = 1, 2, \ldots, 4^{i-1}$ **do**
5: $\quad\quad\quad$ **if** $4^{i-1}j + k > m$ **then** break
6: $\quad\quad\quad$ $[\![\widehat{\delta}_{4^{i-1}j+k}]\!] \leftarrow (j+1)$_MultBit$([\![\widehat{\delta}_k]\!], [\![\widehat{\delta}_{4^{i-1}}]\!])$
7: $\quad\quad$ **end for**
8: $\quad$ **end for**
9: **end for**

Protocol 5 QGuess

Input: $[\![N]\!], [\![D]\!]$
Output: $[\![Q']\!]$, where $Q' \approx \lfloor \frac{N}{D} \rfloor$
1: $[\![\widehat{D'}]\!] \leftarrow \mathsf{ReciprocalGuess}([\![D]\!])$
2: $[\![\widehat{\varepsilon}]\!] \leftarrow -[\![\widehat{D'}]\!] \times [\![D]\!]$
3: $([\![\widehat{\delta}_1]\!], \ldots, [\![\widehat{\delta}_n]\!]) \leftarrow \mathsf{Power}([\![\widehat{\varepsilon}]\!], n)$
4: $[\![\widehat{\delta}]\!] \leftarrow \sum_{i=1}^{n} [\![\widehat{\delta}_i]\!]$
5: $[\![N']\!] \leftarrow 2_\mathsf{MultBit}([\![N]\!], [\![\widehat{D'}]\!])$
6: $[\![Q']\!] \leftarrow [\![N']\!] + 2_\mathsf{MultBit}([\![\widehat{\delta}]\!], [\![N']\!])$

As mentioned in Sect. 3.3, we let the number of inputs M for M_MultBit be at most 4. Note that ε in step 2 of QGuess corresponds to ε in Sect. 3.1, because $\widehat{1 - Y_0 D} = 2^n - 2^n Y_0 D = 2^n - D'D \equiv -D'D \bmod 2^n$.

3.5 Error Analysis

The output of QGuess is less than the exact quotient in general because of rounding errors in M_MultBit. Here, we estimate the size of error by the following lemmas and numerical calculations. In this section, we omit the share symbol $[\![\cdot]\!]$ for short. That is, for example, $z \leftarrow \mathsf{MultBit}(x, y)$ means $z = [\![z]\!]_1 + [\![z]\!]_2 \mod 2^n$ such that $[\![z]\!] \leftarrow \mathsf{MultBit}([\![x]\!], [\![y]\!])$.

Lemma 1. *If the non-zero bits of x, y in binary form are in $x[i]$ $(l_x \leq i \leq u_x)$, $y[j]$ $(l_y \leq j \leq u_y)$, respectively†, then for $z \leftarrow M$_MultBit(x, y), the equations below hold:*

$$x(y2^{-n})^{M-1} - e \leq z \leq x(y2^{-n})^{M-1},$$

where

$$e = \sum_{\substack{(I_1,\ldots,I_{M-1}) \\ \in\{u'_y,\ldots,l'_y\}^{M-1}}} x_0 2^{-s} - (x_0 \gg s),$$

$$u'_y = n - u_y + 1,\ l'_y = n - l_y + 1,$$

†This means that if $x[i] \neq 0$, then $l_x \leq i \leq u_x$ (the same also holds for y). The converse is not assumed.

$$s = \sum_{j=1}^{M-1} I_j,\ x_0 = 2^{u_x} - 2^{l_x - 1}.$$

Here, "≫" means right bit shift. Also, the non-zero bits of z in binary form are in $z[i]$ $(l_z \le i \le u_z)$, where

$$l_z = l_x + (M-1)l_y - (M-1)(n+1),$$
$$u_z = u_x + (M-1)u_y - n(M-1).$$

Proof. Since there does not exist any subtractions in M_MultBit, underflow such as $1 - 3 = 6 \mod 8$ does not occur. Therefore, it is enough to prove the claim for $z' = \sum_{i_1+\cdots+i_{M-1} \le n-1} (x \gg i) \prod_{j=1}^{M-1} y[n - i_j + 1]$ calculated in $\mathbb{Z}$ instead of $\mathbb{Z}_{2^n}$ [†].

First, we define a function f by $f(x,i) = x2^{-i} - (x \gg i)$. Then, the following equation holds (since $x \gg i = 0$ for any $i \ge n$):

$$x(y2^{-n})^{M-1} - z' = \sum_{\substack{(I_1,\ldots,I_{M-1}) \\ \in \{1,\ldots,n\}^{M-1}}} f(x,i) \prod_{j=1}^{M-1} y[n - I_j + 1],$$

where

$$i = \sum_{j=1}^{M-1} I_j.$$

From the assumption about the positions of non-zero bits of x, we have $f(0,i) \le f(x,i) \le f(x_0,i)$. Also, from the assumption about the positions of non-zero bits of y, we can restrict the indices $(I_1,\ldots,I_{M-1})$ of the summation to $\{u'_y,\ldots,l'_y\}^{M-1}$. Hence, we obtain

$$x(y2^{-n})^{M-1} - z' \le \sum_{\substack{(I_1,\ldots,I_{M-1}) \\ \in \{u'_y,\ldots,l'_y\}^{M-1}}} f(x_0,i) \prod_{j=1}^{M-1} 1 = e,$$

and

$$x(y2^{-n})^{M-1} - z' \ge \sum_{\substack{(I_1,\ldots,I_{M-1}) \\ \in \{u'_y,\ldots,l'_y\}^{M-1}}} f(0,i) \prod_{j=1}^{M-1} 0 = 0.$$

Therefore, the inequality holds.

Then, we consider the possible positions of non-zero bits of z'. The least significant non-zero bit of z' has to be at some position to which some non-zero bit of x can be right-shifted in the expression of z'. The size of bit-shift is at most $(n+1-l_y)(M-1)$, so we can let l_z be $l_x + (M-1)l_y - (M-1)(n+1)$. Also, z' is smaller than $2^{u_x+(M-1)(u_y-n)}$ from the inequality above and $x < 2^{u_x}$, $y < 2^{u_y}$. Therefore, non-zero bits of z' are at most $\{u_x + (M-1)u_y - n(M-1)\}$-th bit, and we can let u_z be $u_x + (M-1)u_y - n(M-1)$. □

[†]If the inequality holds for z', then $z' \le x(y2^{-n})^{M-1} < x < 2^n$, and $z = (z' \mod 2^n) = z'$.

Corollary 1. *For l_y, u_y, l'_y, u'_y defined in Lemma 1, if $u_y - l_y + 1 = m$, then we have*

$$xy2^{-n} - m \le z \le xy2^{-n},\ \textit{where } z \leftarrow \mathsf{MultBit}(x,y).$$

Proof. By Lemma 1 (and the proof of Lemma 1), we obtain

$$xy2^{-n} - \sum_{I=u'_y}^{l'_y} f(x,I) \le z \le xy2^{-n}.$$

Since $f(x,I) = x2^{-I} - x \gg I$ is less than 1, we have

$$\sum_{I=u'_y}^{l'_y} f(x,I) < \sum_{I=u'_y}^{l'_y} 1 = m.$$

Hence, the inequality in the claim hold. □

Lemma 2. *For non-negative values a,b,x,y , let x',y',z,z' be*

$$M \in \{2,\ 3,\ 4\},$$
$$\max\{0, x-a\} \le x' \le x,\ \max\{0, y-b\} \le y' \le y,$$
$$z = x(y2^{-n})^{M-1},\ z' = x'(y'2^{-n})^{M-1},$$

then we have

$$\max\{z - c, 0\} \le z' \le z,$$

where

$$c = (M-1)xy^{M-2}2^{-(M-1)n}b + (y2^{-n})^{M-1}a.$$

Proof. $z' \le z$ and $0 \le z'$ are obvious. We prove $z - c \le z'$. First, in the case of $x - a \ge 0,\ y - b \ge 0$, we can deform the equations as below:

$$\begin{aligned} x'y'^{M-1} &\ge (x-a)(y-b)^{M-1} \\ &= \begin{cases} xy - ay - bx + ab & (M=2) \\ xy^2 - 2(x-a)yb + (x-a)b^2 - ay^2 & (M=3) \\ xy^3 - 3(x-a)y^2b & \\ \quad +3(x-a)yb^2 - (x-a)b^3 - ay^3 & (M=4) \end{cases} \\ &\ge \begin{cases} xy - ay - bx & (M=2) \\ xy^2 - 2xyb - ay^2 & (M=3) \\ xy^3 - 3xy^2b - ay^3 & (M=4) \end{cases} \\ &= xy^{M-1} - c2^{(M-1)n}. \end{aligned}$$

Therefore, the given inequality holds. Here, we used $a \le x$, $b \le y$ in deforming the second line to the third line. In the case of $x - a < 0$ or $y - b < 0$, $xy^{M-1} - c2^{(M-1)n}$ is non-positive and z' is non-negative, so the given inequality also holds. □

Lemma 3. *Assume that D is not a power of 2 and let $\varepsilon = 1 - 2^{-d}D$, where d is the bit length of D. Then, for $(\widehat{\delta_1},\ldots,\widehat{\delta_n}) \leftarrow$*

$\mathsf{Power}(\widehat{\varepsilon}, n)$*, the following inequality holds:*

$$(\widehat{\varepsilon})^i 2^{-(i-1)n} - a_i \le \widehat{\delta}_i \le (\widehat{\varepsilon})^i 2^{-(i-1)n}$$

Here, a_i *is defined inductively as follows:*

$$a_1 = 0, a_i = \frac{(M-1)a_{4^j}}{2^{k+(M-2)4^j}} + \frac{a_k}{2^{(M-1)4^j}} + E_i,$$

where

$$\begin{aligned}
&i = 4^j(M-1) + k \\
&\qquad (j = \lfloor \log_4 i \rfloor,\ 1 \le k \le 4^j,\ M \in \{2,3,4\}), \\
&E_i = \sum_{\substack{(I_1,\ldots,I_{M-1}) \\ \in \{n-u_{4^j}+1,\ldots,n-l_{4^j}+1\}^{M-1}}} x_k 2^{-s} - (x_k \gg s), \\
&s = \sum_{j=1}^{M-1} I_j,\ x_k = 2^{u_k} - 2^{l_k - 1}, \\
&u_k = n - k,\ l_k = \max\{0, n - kd - k + 1\}.
\end{aligned}$$

Proof. First, we consider the range of non-zero bits of $\widehat{\delta}_i$. Since D is not a power of 2 and the bit length of D is equal to d, the non-zero bits of $\widehat{\delta}_1 = \widehat{\varepsilon}$ is in $\widehat{\delta}_1[m]$ $(l_1 \le m \le u_1)$, where $l_1 = n - d + 1$, $u_1 = n - 1$. Also, by Lemma 1, we obtain for $i = 4^j(M-1) + k$,

$$\begin{aligned}
l_i &= l_k + (M-1)l_{4^j} - (M-1)(n+1), \\
u_i &= u_k + (M-1)l_{4^j} - n(M-1).
\end{aligned}$$

By induction, we get that the range of non-zero bits of $\widehat{\delta}_i$ is in $\widehat{\delta}_i[m]$ $(l_i \le m \le u_i)$, where $l_i = n - id - i + 1$, $u_i = n - i$[†]. Especially, we can let $l_i = \max\{0, n - id - i + 1\}$.

Now, we prove the inequality in the claim by induction. First, in the case of $i = 1$, the inequality is obvious. It is enough to prove the inequality for $i = 4^j(M-1) + k$, $(M = 2,3,4, k = 1,2,\ldots,4^j)$ on the assumption that the inequality holds for $i \le 4^j$. by Lemma 1, we obtain

$$\widehat{\delta}_k(\widehat{\delta}_{4^j} 2^{-n})^{M-1} - E_i \le \widehat{\delta}_i \le \widehat{\delta}_k(\widehat{\delta}_{4^j} 2^{-n})^{M-1}.$$

From the assumption of induction and the fact that $\widehat{\varepsilon} < 2^{n-1}$, we get

$$\begin{aligned}
&(\widehat{\varepsilon})^k 2^{-(k-1)n} - a_k \le \widehat{\delta}_k \le (\widehat{\varepsilon})^k 2^{-(k-1)n}, \\
&(\widehat{\varepsilon})^{4^j} 2^{-(4^j-1)n} - a_{4^j} \le \widehat{\delta}_{4^j} \le (\widehat{\varepsilon})^{4^j} 2^{-(4^j-1)n}.
\end{aligned}$$

Therefore, by Lemma 2, we obtain

$$(\widehat{\varepsilon})^i 2^{-(i-1)n} - c' \le \widehat{\delta}_k(\widehat{\delta}_{4^j} 2^{-n})^{M-1} \le (\widehat{\varepsilon})^i 2^{-(i-1)n},$$

where,

$$\begin{aligned}
c' =& (M-1)(\widehat{\varepsilon})^k 2^{-(k-1)n} \{(\widehat{\varepsilon})^{4^j} 2^{-(4^j-1)n}\}^{M-2} 2^{-(M-1)n} a_{4^j} \\
&+ \{(\widehat{\varepsilon})^{4^j} 2^{-(4^j-1)n} 2^{-n}\}^{M-1} a_k.
\end{aligned}$$

[†]In fact, $(n-kd-k+1)+(M-1)(n-4^j d-4^j+1)-(M-1)(n+1) = n - id - i + 1$ and $(n-k) + (M-1)(n-4^j) - n(M-1) = n - i$.

Here, since $\widehat{\varepsilon} 2^{-n} \le \frac{1}{2}$, we get

$$\begin{aligned}
c' &\le (M-1)2^{-k} 2^n (2^{-4^j} 2^n)^{M-2} 2^{-(M-1)n} a_{4^j} + 2^{-4^j(M-1)} a_k \\
&= \frac{(M-1)a_{4^j}}{2^{k+(M-2)4^j}} + \frac{a_k}{2^{(M-1)4^j}}
\end{aligned}$$

From these inequalities, we obtain the inequality in the claim. □

Lemma 4. *Assume that* $\varepsilon = 1 - 2^{-d}D$*, where* d *is the bit length of* D*. If* $0 \le (\sum_{i=1}^{n} (\widehat{\varepsilon})^i 2^{-(i-1)n}) - \widehat{\delta} \le E$*, then we have*

$$\lfloor \frac{N}{D} - \frac{5}{2} - 2^{-d}E - n \rfloor \le Q' \le \lfloor \frac{N}{D} \rfloor,$$

where

$$Q' = N' + \mathsf{MultBit}(N', \widehat{\delta}), N' = \mathsf{MultBit}(N, 2^{n-d}).$$

Proof. Since the number of non-zero bits of 2^{n-d} is equal to one (the $(n-d+1)$-th least bit), we have

$$N2^{-d} - 1 \le N' \le N2^{-d}$$

by Corollary 1. Also, we have

$$N'\widehat{\delta}2^{-n} - n \le \mathsf{MultBit}(N', \widehat{\delta}) \le N'\widehat{\delta}2^{-n}$$

since the number of non-zero bits is at most n. From these two inequalities and the assumption about $\widehat{\delta}$, we get

$$\begin{aligned}
Q' &\le N2^{-d} + N2^{-d}\widehat{\delta}2^{-n} \le N2^{-d} + N2^{-d} \sum_{i=1}^{n} \varepsilon^i \\
&= N2^{-d} \frac{1 - \varepsilon^{n+1}}{1 - \varepsilon} < N2^{-d} \frac{1}{1 - \varepsilon} = \frac{N}{D}.
\end{aligned}$$

Also, we obtain

$$\begin{aligned}
Q' &\ge N2^{-d} - 1 + N' \sum_{i=1}^{n} \varepsilon^i - N'E2^{-n} - n \\
&\ge N2^{-d} - 1 + (N2^{-d} - 1) \sum_{i=1}^{n} \varepsilon^i - N2^{-d}E2^{-n} - n \\
&= \frac{N}{D} - \frac{N}{D}\varepsilon^{n+1} - \frac{1 - \varepsilon^{n+1}}{1 - \varepsilon} - \frac{N}{2^n} 2^{-d} E - n \\
&\ge \frac{N}{D} - \frac{1}{2} - 2 - 2^{-d}E - n.
\end{aligned}$$

Hence, by applying the floor function, we have the inequality in the claim. □

Using Lemma 3, we can compute a_i recursively by a numerical experiment. Let n be 64, E be $\sum_{i=1}^{64} a_i$, and d be the bit length of D. (Note that this E corresponds to E in Lemma 4.) Then we got $2^{-d}E < 24$ for $d \ge 12$ by the experiment above. Also, for $d \le 11$ and $D = 2^i (i = 1,\ldots,64)$, we got $2^{-d}E < 40.5$. Therefore, with Lemma 4, $Q' \leftarrow \mathsf{QGuess}(N, D)$ satisfies $\lfloor \frac{N}{D} \rfloor - 107 < Q' \le \lfloor \frac{N}{D} \rfloor$. By conducting a similar experiment for 32-bit integers ($n = 32$), we obtain $\lfloor \frac{N}{D} \rfloor - 54 < Q' \le \lfloor \frac{N}{D} \rfloor$.

Protocol 6 ErrorCorrect

Input: $[\![N]\!], [\![D]\!], [\![Q']\!], [\![A]\!]$, where $\lfloor \frac{N}{D} \rfloor \in \{Q', Q'+1, \ldots, Q'+A-1\}$
Output: $[\![Q]\!]$, where $Q = \lfloor \frac{N}{D} \rfloor$
1: $[\![N']\!] \leftarrow [\![N]\!] - [\![Q']\!] \times [\![D]\!]$
2: $[\![\delta]\!] \leftarrow$ Equal_zero$([\![Q']\!])$
3: **for** $i = 1, 2, \ldots, A$ **do**
4: $\quad [\![b_i]\!] \leftarrow$ Comparison$([\![N']\!], i \times [\![D]\!])$
5: **end for**
6: $([\![b'_1]\!], \ldots, [\![b'_A]\!]) \leftarrow$ MSNZB$([\![b_1]\!], \ldots, [\![b_A]\!])$
7: $[\![q]\!] \leftarrow \sum_{i=1}^{A} (i-1) \times [\![b'_i]\!]$
8: $[\![Q]\!] \leftarrow [\![\delta]\!] \times ([\![1]\!] - [\![b_1]\!]) + ([\![1]\!] - [\![\delta]\!]) \times ([\![Q']\!] + [\![q]\!])$

3.6 Correction of Rounding Errors – ErrorCorrect

In [12], the correctness of the protocol was guaranteed by adding a correction term Δ for canceling rounding errors. However, rounding errors in the case using M_MultBit is larger than in [12], and it seems difficult to find out the explicit correction term. Here, we construct ErrorCorrect protocol which computes the exact quotient known to be in a given range.

We assume that the exact quotient is in $\{Q', Q'+1, \ldots, Q'+A-1\}$. The idea of ErrorCorrect is that we compute $N \overset{?}{<} Q'D, N \overset{?}{<} (Q'+1)D, \ldots, N \overset{?}{<} (Q'+A-1)D$ and find out the first position of False. However, if we do it naively, we cannot find out the precise position in the case of $\lfloor \frac{N}{D} \rfloor D \le N < 2^n \le (\lfloor \frac{N}{D} \rfloor + 1)D$[†].

Here, we can avoid this problem by the following lemma:

Lemma 5. *If $Q' = 0$, the exact quotient is equal to 0 or 1.*

Proof. If $Q' = 0$, then N' in step 5 of QGuess is equal to 0 and this means that the bit length of N is at most d, where d is the bit length of D. Therefore, N is less than $2D$ and $\lfloor \frac{N}{D} \rfloor$ is equal to 0 or 1. □

Using this lemma, we can compute the exact quotient by comparing $D, 2D, \ldots, (A-1)D$ with $N - Q'D$ and comparing D with N in parallel, and judging whether Q' is equal to 0 or not. In the case of $Q' = 0$, $\frac{N}{D} = 0$ or 1 by the lemma, and this can be computed by comparing D and N. In the case of $Q' \ge 1$, the problem described above does not occur. This is because the problem occurs in the case of $(\lfloor \frac{N}{D} \rfloor - Q')D \le N - Q'D < 2^n \le (\lfloor \frac{N}{D} \rfloor - Q' + 1)D$ and this never holds if $Q' \ge 1$. More specifically, if $Q' \ge 1$ and $(\lfloor \frac{N}{D} \rfloor - Q')D \le N - Q'D$, then $(\lfloor \frac{N}{D} \rfloor - Q' + 1)D \le N - (Q'-1)D \le N < 2^n$. Therefore, $(\lfloor \frac{N}{D} \rfloor - Q')D \le N - Q'D$ and $2^n \le (\lfloor \frac{N}{D} \rfloor - Q' + 1)D$ never holds at the same time.

The resulting protocol is given in Protocol 6.

3.6.1 Correctness

In the case of $Q' = 0$, $\lfloor \frac{N}{D} \rfloor$ is equal to the value corresponding to $1 - (N \overset{?}{<} D)$ from the assumption. In the other case, the minimum index i such that $N' < iD$ is equal to $q+1$[††]. Therefore, $qD \le N' < (q+1)D$ and $\lfloor \frac{N}{D} \rfloor = Q' + q$ holds. Summarizing the two cases above, $\lfloor \frac{N}{D} \rfloor = \delta(1 - b_1) + (1 - \delta)(Q' + q)$ holds.

[†]Since we treat integers as elements of $\mathbb{Z}_{2^n}$, in the case above, $(\lfloor \frac{N}{D} \rfloor + 1)D$ is equal to $(\lfloor \frac{N}{D} \rfloor + 1)D - 2^n$ and less than N.

Protocol 7 Division

Input: $[\![N]\!], [\![D]\!]$
Output: $[\![Q]\!]$, where $Q = \lfloor \frac{N}{D} \rfloor$
1: $[\![Q']\!] \leftarrow$ QGuess$([\![N]\!], [\![D]\!])$
2: $[\![Q]\!] \leftarrow$ ErrorCorrect$([\![N]\!], [\![D]\!], [\![Q']\!], A)$

3.7 Summary of Division Protocol

With QGuess and ErrorCorrect, we can compute our division protocol (Protocol 7). As discussed in Sect. 3.5, we can set $A = 54$ for 32-bit integers and $A = 107$ for 64-bit integers.

3.8 Division for Fixed-Point Numbers

The division protocol described above can be applied to division for fixed point numbers. Let N, D be fixed point numbers with f bit precision. That is, $N = \bar{N} \times 2^{-f}, D = \bar{D} \times 2^{-f}$, where $\bar{N}, \bar{D} \in \mathbb{Z}_{2^n}$. In this case, the fixed point representation (with f bit precision) of $Q = \frac{N}{D}$ can be expressed as $\bar{Q} = \lfloor \frac{\bar{N} \times 2^f}{\bar{D}} \rfloor \in \mathbb{Z}_{2^n}$. One may think that bit expansion is needed since $\bar{N} \times 2^f$ does not necessarily belong to Z_{2^n}. However, we can avoid this problem as follows: Let d be the bit length of $\bar{D}$ and $\hat{\delta}$ be defined as step 4 of QGuess with input $\bar{D}$[†††]. Then, the following approximation is hold as in QGuess:

$$\begin{aligned}\bar{Q} &\approx \bar{N} \times 2^{f-d} + \bar{N} \times \hat{\delta} \times 2^{f-d-n} \\ &\approx \bar{N} \times 2^{f-d} + \bar{N}' \times \hat{\delta} \times 2^{f-n},\end{aligned}$$

where $\bar{N}' = \bar{N} \times 2^{-d}$. (Note that the approximation above is equal to the output of QGuess in the case of $f = 0$.) Though 2_MultBit(x, y) computes an approximate value of $xy2^{-n}$, we can easily extend it to compute an approximate value of $xy2^{f-n}$ (without bit expansion). Therefore, $\bar{Q}$ can be computed similarly to QGuess, and error analysis can be done in the same way.

4. Constant-Round Building Blocks

In this section, we give a constant-round construction of the protocol Overflow used in RightShift, ExtractBit, and Comparison. Overflow receives two inputs $([\![x]\!], t)$, and computes a share (over $\mathbb{Z}_{2^n}$) of the boolean value corresponding to whether $[\![x]\!]_1[t \ldots 1] + [\![x]\!]_2[t \ldots 1] \ge 2^t$ or not.

4.1 List of Subprotocols

Here, we introduce subprotocols used in Overflow. Each

[††]From the assumption that the exact quotient is in $\{Q', Q'+1, \ldots, Q'+A-1\}$, $N' \ge 0$ and $N' < iD$ holds for some indexes i.
[†††]Note that $\hat{\delta}$ depends only D in QGuess.

subprotocol except for assump_Overflow deals with shares over $\mathbb{F}_p$, where p is an odd prime satisfying $n \le p < \sqrt{2^n}$. Note that these subprotocols can be easily implemented without using integers larger than n-bit. The input of assump_Overflow is a share over $\mathbb{F}_p$, and the output is a share over $\mathbb{Z}_{2^n}$. To make it easier to understand, we use a symbol $[\![\cdot]\!]^{\langle p\rangle}$ for a share over $\mathbb{F}_p$.

- Pow : $[\![y]\!]^{\langle p\rangle} \leftarrow \mathsf{Pow}([\![x]\!]^{\langle p\rangle}, k)$, where y is equal to x^k[†].
- Equal_one : $[\![y]\!]^{\langle p\rangle} \leftarrow \mathsf{Equal_one}([\![x]\!]^{\langle p\rangle})$, where y is the boolean value corresponding to $x \stackrel{?}{=} 1$ on the assumption $0 \le x \le n$.
- assump_Overflow : $[\![y]\!] \leftarrow \mathsf{assump_Overflow}([\![x]\!]^{\langle p\rangle})$, where y is the boolean value corresponding to $[\![x]\!]_1^{\langle p\rangle} + [\![x]\!]_2^{\langle p\rangle} \stackrel{?}{\ge} p$ on the assumption $x < \frac{p}{2}$.

4.2 Pow

By using $([\![a]\!]^{\langle p\rangle}, [\![a^2]\!]^{\langle p\rangle}, \ldots, [\![a^k]\!]^{\langle p\rangle})$ as auxiliary inputs (with random and unknown value a) instead of standard Beaver triplet, we can securely compute Pow with one round. First, each party computes $[\![x-a]\!]^{\langle p\rangle}$ and then gets $x' = x - a$ by using Reconst. Second, party 1 computes $x'^k + \sum_{i=0}^{k-1} \binom{k}{i} x'^i [\![a^{k-i}]\!]_1^{\langle p\rangle}$ and party 2 computes $\sum_{i=0}^{k-1} \binom{k}{i} x'^i [\![a^{k-i}]\!]_2^{\langle p\rangle}$. Since

$$\begin{aligned} x^k = (x'+a)^k &= \sum_{i=0}^{k} \binom{k}{i} x'^i a^{k-i} \\ &= \binom{k}{k} x'^k + \sum_{i=0}^{k-1} \binom{k}{i} x'^i ([\![a^{k-i}]\!]_1^{\langle p\rangle} + [\![a^{k-i}]\!]_2^{\langle p\rangle}) \\ &= \left\{ x'^k + \sum_{i=0}^{k-1} \binom{k}{i} x'^i [\![a^{k-i}]\!]_1^{\langle p\rangle} \right\} \\ &\quad + \left\{ \sum_{i=0}^{k-1} \binom{k}{i} x'^i [\![a^{k-i}]\!]_2^{\langle p\rangle} \right\}, \end{aligned}$$

these values are valid shares of x^k. (Note that the equation above is over $\mathbb{F}_p$.)

4.3 Equal_one

This function was constructed in [24] using Fermat's little theorem. That is, $f(x) := 1 - x^{p-1}$ is equal to 1 at $x = 0$ and equal to 0 at other points. Therefore, $f(x-1)$ can be regarded as the boolean value corresponding to $x \stackrel{?}{=} 1$. For computing $f(x-1)$, it is enough to compute $(x-1)^{p-1}$, and this can be done with Pow with one round.

4.4 assump_Overflow

If $x < \frac{p}{2}$, then

$$[\![x]\!]_1^{\langle p\rangle} + [\![x]\!]_2^{\langle p\rangle} < p \Leftrightarrow [\![x]\!]_1^{\langle p\rangle} < \frac{p}{2} \ \wedge \ [\![x]\!]_2^{\langle p\rangle} < \frac{p}{2}.$$

The right side is the product of r_1 and r_2, where r_i is the boolean value corresponding to $[\![x]\!]_i^{\langle p\rangle} \stackrel{?}{<} \frac{p}{2}$ which can be computed locally. Since the negation can be computed locally, we can compute assump_Overflow with one round.

4.5 Overflow

From now, we construct Overflow with subprotocols above. We show some examples at the end of this section.

Here, we define the array $\mathbf{X}$ whose length is n by $\mathbf{X}[i] = [\![x]\!]_1[i] + [\![x]\!]_2[i]$ $(i = 1, 2, \ldots, n)$. For example, in the case of $n = 3$, $x = 5$, $[\![x]\!]_1 = 6$, $[\![x]\!]_2 = 7$, $\mathbf{X}$ is (2,2,1)[††].

In this setting, $[\![x]\!]_1[t \ldots 1] + [\![x]\!]_2[t \ldots 1] \ge 2^t$ if and only if:

> There exists an element 2 in $(\mathbf{X}[t], \ldots, \mathbf{X}[1])$ and the leftmost non-one element of $(\mathbf{X}[t], \ldots, \mathbf{X}[1])$ is equal to 2.

This fact can be understood by considering that carrying up to the $(t+1)$-th digit occurs if and only if $\mathbf{X}[t] = 2$ or "$\mathbf{X}[t] = 1$ and there is carry-up at $(t-1)$-th digit".

The important point is, since each element of $\mathbf{X}$ is in $\{0, 1, 2\}$ and each share of $\mathbf{X}[i]$ (that is, $[\![x]\!][i]$) is in $\{0, 1\}$, we can regard $([\![x]\!]_1[i], [\![x]\!]_2[i])$ as a share of $\mathbf{X}[i]$ over $\mathbb{F}_p$. Based on this fact, we can apply operations over $\mathbb{F}_p$. We set $t = n$ for simplicity.

4.5.1 First Step

We apply a function, which maps 0, 2 to 1 and 1 to 0, to each element of $\mathbf{X}$. That is, we compute array $\mathbf{Y}$ as follows; $\mathbf{Y}[i] = (\mathbf{X}[i] - 1)^2$ $(i = 1, \ldots, t)$. This can be computed with one round. In parallel, we also compute array $\mathbf{Y}'$ by applying a function which maps 2 to 1 and 0, 1 to 0 to each element of $\mathbf{X}$; $\mathbf{Y}'[i] = \frac{\mathbf{X}[i](\mathbf{X}[i]-1)}{2}$ $(i = 1, \ldots, t)$. Here, "/2" means the division by 2 over the field $\mathbb{F}_p$ and this can be computed locally since this operation is equivalent to constant multiplication.

4.5.2 Second Step

We find out whether the leftmost position (that is, the nearest to t-th position) of 1 in $\mathbf{Y}$ corresponds to 2 in $\mathbf{X}$ or not. First, we compute reverse cumulative sum of $\mathbf{Y}$. That is, we compute array $\mathbf{Z}$ as follows; $\mathbf{Z}[i] = \sum_{k=i}^{t} \mathbf{Y}[k]$ $(i = 1, \ldots, t)$.

This can be computed locally. Second, we compute array $\mathbf{Z}'$ by applying Equal_one to each element of $\mathbf{Z}$; $\mathbf{Z}'[i] \leftarrow \mathsf{Equal_one}(\mathbf{Z}[i])$. Note that any element of $\mathbf{Z}$ is non-negative and at most $t \le n \le p$. Finally, we compute the inner product s of $\mathbf{Z}'$ and $\mathbf{Y}'$. It takes one round to compute $\mathbf{Z}'$, and takes one round to compute the inner product.

[†]Though we treat Pow only over $\mathbb{F}_p$, we can construct Pow over $\mathbb{Z}_{2^n}$ similarly.

[††]Matching with binary expression, the rightmost component of $\mathbf{X}$ corresponds to $\mathbf{X}[1]$.

Protocol 8 Overflow

Input: $[\![x]\!], t$

Output: $[\![s]\!]$, where s=$([\![x]\!]_1[t\ldots1] + [\![x]\!]_2[t\ldots1] \overset{?}{\geq} 2^t)$

1: $[\![\mathbf{Y}[i]]\!]^{\langle p\rangle} \leftarrow ([\![\mathbf{X}[i]-1]\!]^{\langle p\rangle}) \times ([\![\mathbf{X}[i]-1]\!]^{\langle p\rangle})$ for $i = 1,\ldots,t$.
2: $[\![\mathbf{Y}'[i]]\!]^{\langle p\rangle} \leftarrow \frac{[\![\mathbf{X}[i]]\!]^{\langle p\rangle} \times ([\![\mathbf{X}[i]-1]\!]^{\langle p\rangle})}{2}$ for $i = 1,\ldots,t$.
3: $[\![\mathbf{Z}[i]]\!]^{\langle p\rangle} \leftarrow \sum_{k=i}^{t} [\![\mathbf{Y}[k]]\!]^{\langle p\rangle}$ for $i = 1,\ldots,t$.
4: $[\![\mathbf{Z}'[i]]\!]^{\langle p\rangle} \leftarrow$ Equal_one$([\![\mathbf{Z}[i]]\!]^{\langle p\rangle})$ for $i = 1,\ldots,t$.
5: $[\![s]\!]^{\langle p\rangle} \leftarrow \sum_{i=1}^{t} [\![\mathbf{Z}'[i]]\!]^{\langle p\rangle} \times [\![\mathbf{Y}'[i]]\!]^{\langle p\rangle}$
6: $[\![s]\!] \leftarrow \mathsf{CastUp}_{p\to 2^n}([\![s]\!]^{\langle p\rangle})$

4.5.3 Third Step

Since the output computed above is a share over $\mathbb{F}_p$, we need to transform it to a share over $\mathbb{Z}_{2^n}$. We do this using $\mathsf{CastUp}_{p\to 2^n}$ which can be constructed in the similar way to $\mathsf{CastUp}_{2^n\to 2^m}$. Also, in this case, we can replace Overflow in CastUp [12] with assump_Overflow. That is, s is equal to $[\![s]\!]_1^{\langle p\rangle} + [\![s]\!]_2^{\langle p\rangle} - p \times b$, where b is the boolean value corresponding to $[\![s]\!]_1^{\langle p\rangle} + [\![s]\!]_2^{\langle p\rangle} \overset{?}{\geq} p$, and $[\![b]\!]$ can be computed by assump_Overflow. Note that we can use assump_Overflow because s is equal to 0 or 1 and less than $p/2$.

4.5.4 Summary of Overflow Protocol and Other Building Blocks

The summary of Overflow is given in Protocol 8. Note that array $\mathbf{X}$ in step 1 is defined at the beginning of Sect. 4.5. It takes one round in step 1,2 in parallel, and one round in step 4-6, respectively. Therefore, Overflow can be computed with four rounds in total. Furthermore, by using $([\![a]\!]^{\langle p\rangle},\ldots,[\![a^k]\!]^{\langle p\rangle},[\![b]\!]^{\langle p\rangle},[\![ba]\!]^{\langle p\rangle},\ldots,[\![ba^k]\!]^{\langle p\rangle})$ as auxiliary inputs, we can compute step 4,5 together with one round in the similar way to Pow. Hence, we can compute Overflow with three rounds.

Using this Overflow, RightShift and ExtractBit can be computed in three rounds and Comparison can be computed in four rounds. Also, Equal_zero and MSNZB can be computed in three rounds in the similar way to Overflow. We give the detail description of these protocols in Sect. 4.6.

4.5.5 Example

We show an example of Overflow. Each symbol corresponds to ones in summary of Overflow. We let $n = t = 8$. $(\cdot)_2$ means the binary notation.

$$
\begin{aligned}
&x = 42,\\
&[\![x]\!]_1 = 126 = (01111110)_2,\\
&[\![x]\!]_2 = 172 = (10101100)_2,\\
&\mathbf{X} = (1,1,2,1,2,2,1,0),\\
&\mathbf{Y} = (0,0,1,0,1,1,0,1),\ \mathbf{Y}' = (0,0,1,0,1,1,0,0),\\
&\mathbf{Z} = (0,0,1,1,2,3,3,4),\ \mathbf{Z}' = (0,0,1,1,0,0,0,0),\\
&s = 1.
\end{aligned}
$$

4.6 Other Building Blocks

In this section, we show the construction of RightShift, ExtractBit, and Comparison using Overflow. (See [22] for more detail.) Also, we give the construction of Equal_zero and MSNZB in the similar way to Overflow.

4.6.1 RightShift, ExtractBit

Since Overflow compute the carry bit of $[\![x]\!]_1 + [\![x]\!]_2$, RightShift and ExtractBit can be computed using Overflow as following:

$$
\begin{aligned}
\mathsf{RightShift}([\![x]\!],t) &= [\![x]\!] \gg t + \mathsf{Overflow}([\![x]\!],t)\\
&\quad - 2^{n-t} \cdot \mathsf{Overflow}([\![x]\!],n),\\
\mathsf{ExtractBit}([\![x]\!],t) &= [\![x]\!][t] + \mathsf{Overflow}([\![x]\!],t-1)\\
&\quad - 2 \cdot \mathsf{Overflow}([\![x]\!],t).
\end{aligned}
$$

Here, $x \gg t$ means the t bits right shift of x (i.e., $x \gg t = \lfloor \frac{x}{2^t} \rfloor$). The number of rounds of these protocol are the same as that of Overflow.

4.6.2 Comparison

Comparison protocol can be computed using ExtractBit. Let a, b, and c denote the most significant bit of x, y, and $x - y$. In the case of $a \oplus b = 1$, either one of x and y is less than 2^{n-1} and the other is greater than or equal to 2^{n-1}. Therefore, $x \overset{?}{<} y$ is equal to the most significant bit of y, that is, b. In the case of $a \oplus b = 0$, $x \overset{?}{<} y$ is equal to the most significant bit of $x - y$, that is, c, since $(x-y) \bmod 2^n \geq 2^{n-1} \Leftrightarrow x < y$ when the most significant bit of x and y are the same. In summary, the following holds:

$$
\begin{aligned}
x \overset{?}{<} y &= (a \oplus b)b + (a \oplus b \oplus 1)c\\
&= (a-b)^2 b + \{1-(a-b)^2\}c\\
&= (a-b)^2(b-c) + c.
\end{aligned}
$$

Therefore, we can compute Comparison by (i) computing $[\![a]\!]$, $[\![b]\!]$, $[\![c]\!]$ with ExtractBit and (ii) computing $[\![a-b]\!]^2 \times [\![b-c]\!] + [\![c]\!]$ with the extension of Pow as mentioned in Sect. 4.5.4. It needs three rounds in (i) and one round in (ii), and four rounds in total.

4.6.3 Equal_zero

In this section, we construct Equal_Zero protocol in a similar way to Overflow. First, P_2 computes $x' = -[\![x]\!]_2 \bmod 2^n$. Then, $x = 0$ holds if and only if $[\![x]\!]_1 = x'$ holds. Let $\mathbf{X}$ be an array whose length is n and $\mathbf{X}[i] = [\![x]\!]_1[i] + x'[i]$. As in Overflow, $([\![x]\!]_1[i], x'[i])$ can be regarded as a share of $\mathbf{X}[i]$ over $\mathbb{F}_p$. Also, the condition $[\![x]\!]_1 = x'$ holds if and only if $X[i]$ is equal to 0 or 2 for all i. Therefore, we can compute Equal_Zero by computing $[\![\mathbf{Y}[i]]\!]^{\langle p\rangle} \leftarrow 1 - [\![\mathbf{X}[i]-1]\!]^{\langle p\rangle} \times$

Table 1 Execution time of Division.

	pre comp [ms]	online comp [ms]	data trans [KB]	comm round	est comm time [ms]
32-bit	6.2	1.65	71.8	31	2240
64-bit	21.9	11.6	310	31	2266

Protocol 9 Equal_Zero

Input: $[\![x]\!]$

Output: $[\![s]\!]$, where s=$(x \stackrel{?}{=} 0)$

1: $[\![\mathbf{Y}[i]]\!]^{\langle p\rangle} \leftarrow 1 - ([\![\mathbf{X}[i] - 1]\!]^{\langle p\rangle}) \times ([\![\mathbf{X}[i] - 1]\!]^{\langle p\rangle})$ for $i = 1, \ldots, n$.
2: $[\![z]\!]^{\langle p\rangle} = \sum_{i=1}^{n} [\![Y[i]]\!]^{\langle p\rangle}$
3: $[\![s]\!]^{\langle p\rangle} \leftarrow$ Equal_one$([\![z + 1]\!]^{\langle p\rangle})$
4: $[\![s]\!] \leftarrow$ CastUp$_{p\rightarrow 2^n}([\![s]\!]^{\langle p\rangle})$

Protocol 10 MSNZB

Input: $[\![x_1]\!], \ldots, [\![x_n]\!]$, where each x_i is equal to 0 or 1.

Output: $([\![y_1]\!], \ldots, [\![y_n]\!])$, where $y_i = 1$ in the case that $x_i = 1$ and $x_j = 0$ for all $j < i$ and $y_i = 0$ otherwise.

1: $[\![\mathbf{Y}[i]]\!]^{\langle p\rangle} \leftarrow [\![x_i]\!] \bmod p - 2^n$assump_Overflow$([\![x_i]\!])$ for $i = 1, \ldots, n$.
2: $[\![\mathbf{Z}[i]]\!]^{\langle p\rangle} \leftarrow \sum_{k=1}^{i} [\![\mathbf{Y}[k]]\!]^{\langle p\rangle}$ for $i = 1, \ldots, n$.
3: $[\![\mathbf{Z}'[i]]\!]^{\langle p\rangle} \leftarrow$ Equal_one$([\![\mathbf{Z}[i]]\!]^{\langle p\rangle})$ for $i = 1, \ldots, n$.
4: $[\![y_i]\!]^{\langle p\rangle} \leftarrow [\![\mathbf{Z}'[i]]\!]^{\langle p\rangle} \times [\![Y[i]]\!]^{\langle p\rangle}$ for $i = 1, \ldots, n$.
5: $[\![y_i]\!] \leftarrow$ CastUp$_{p\rightarrow 2^n}([\![y_i]\!]^{\langle p\rangle})$ for $i = 1, \ldots, n$.

$[\![\mathbf{X}[i] - 1]\!]^{\langle p\rangle}$, summing up $[\![y]\!]\langle p\rangle = \sum_{i=1}^{n} [\![\mathbf{Y}[i]]\!]^{\langle p\rangle}$, and checking $y + 1$ is equal to 1 or not using Equal_one. The detailed description of Equal_Zero is given in Protocol 9. It needs one round in each step 1, step 2, and step 4. Therefore, the number of rounds of Equal_Zero is three rounds in total.

4.6.4 MSNZB

In this section, we construct MSNZB protocol in a similar way to Overflow. MSNZB protocol can be understood as a protocol that detects which is the leftmost non-zero bit. As mentioned at the beggining of Sect. 4.5, we compute the leftmost not-one element of an array $\mathbf{X}$ and check it is equal to 2 or not in Overflow protocol. Therefore, we can detect the leftmost non-zero bit in a similar way. Fisrt, we convert the the input to the share over $\mathbb{F}_p$. This can be computed using assump_Overflow. Note that the input of assump_Overflow is a share over $\mathbb{F}_1$, we can extend this to a share over $\mathbb{Z}_{2^n}$ with the same construction. Then, we do the same process as in steps 3-6 of Protocol 8. We give the detailed description of MSNZB in Protocol 10.

It needs one round in step 1, one round in step 3-4, and one round in step 5. Therefore, the number of rounds of MSNZB is three in total.

4.7 Comparison with Related Works

To the best of our knowledge, our Overflow (and its extensions) are the first constant-round secure protocols that work not over $\mathbb{F}_p$ but over $\mathbb{Z}_{2^n}$. Moreover, in comparison with the state of the art constant-round Comparison protocol in two-party setting [17], our Comparison protocol is better in terms of communication rounds and data transfer. In fact, [17] needs five rounds and $O((\log q)^3)$ bit data transfer for SS over $\mathbb{F}_q$, while our protocol needs four rounds and $O(n \log p)$ bit data transfer. Since our protocol is over $\mathbb{Z}_{2^n}$, we can set $q \approx 2^n$. Moreover, as described at the beginning of this section, $p \approx n$. In summary, our protocol significantly improve data transfer ($O(n^3) \rightarrow O(n \log n)$).

5. Evaluations of Efficiency

5.1 Round Complexity

In M_MultBit, we compute RightShift and ExtractBit in parallel, and compute a product of M numbers. Therefore, M_MultBit takes $3 + \lceil \log_2 M \rceil$ rounds[†]. The number of communication rounds in Power is equal to $\lceil \log_4 n \rceil$ times the number of communication rounds in 4_MultBit. Taking into consideration that step 5 in QGuess can be computed in parallel with step 3, we can compute QGuess in $9 + 5\lceil \log_4 n \rceil$ rounds. Also, taking into consideration that step 2 in ErrorCorrect can be computed in parallel with step 3-6, we can compute ErrorCorrect in nine rounds. Furthermore, by not applying CastUp$_{p\rightarrow 2^n}$ in the last step of Comparison and treating the output of Comparison in step 4 and the input of MSNZB in step 6 as shares over $\mathbb{F}_p$, we can compute ErrorCorrect in seven rounds. In total, we can compute Division in 31 rounds for $n = 32, 64$.

5.2 Data Transfer and Execution Time

We implemented Division protocol in C++ programming language. We use a single laptop computer (Core i7-6700 4 GHz, 64 GB RAM). Instead of using actual networks, we estimate communication costs according to communication bits and communication rounds. In the pre-computation phase, we use fixed-key AES as a pseudorandom generator. We assume the WAN setting and the bandwidth and network delay are 9MB/s and 72ms, respectively[††].

We use $p = 37$(67, resp.) in QGuess, and use $p = 59$(107, resp.) in ErrorCorrect for 32-bit(64-bit, resp.) integers. Table 1 shows pre-computation time (pre comp), online computation time (online comp), data transfer (data trans), communication rounds (comm round), and estimated communication time (est comm time). The computation time is the average time of 100 times executions. In total, it takes 2246ms, 2357ms to execute Divide for 32-bit, 64-bit integers, respectively.

[†]Note that a product of M numbers can be computed by executing a product of two numbers $\lceil \log_2 M \rceil$ times.

[††]This setting was used in [15].

5.3 Comparison with Related Works

As described in Sect. 1, we mainly focus on SS-based MPC and two-party setting. To the best of our knowledge, [12] is the state of the art in this setting[†]. In Protocol 1 [12], log-round building blocks constructed in [22] were used. Now, we calculate the number of rounds of Protocol 1. In step 1 of Protocol 1, it needed $\lceil \log_2 n \rceil + 3$ rounds. In step 3, step 7, and step 11, it needed 1 rounds for each step. In step 8 and step 12, it needed $\lceil \log_2 m \rceil + 2$ rounds for each step. Step 2 can be computed with step 1 in parallel and does not affect on the number of rounds. Considering the for-loop, it needed $\lceil \log_2 n \rceil + (h_0 + 1)\lceil \log_2 m \rceil + 3h_0 + 7$ rounds in total. Therefore, it needed 69 rounds (87 rounds, resp.) for 32-bit (64-bit, resp.) integer division. As for the data transfer size, it needed $6n^2(\lceil \log_2 n \rceil + 1) + 4n^2 + 2nm$ bits in step1. In step 2, it needed $4n(\lceil \log_2 n \rceil + 1) + 4m$ bits. In step 3, step 7, and step 11, it needed $4m$ bits for each step. In step 8 and step 12, it needed $4m(\lceil \log_2 m \rceil + 1) + 4m$ bits for each step. Considering the for-loop, it needed $(6n^2 + 4n)(\lceil \log_2 n \rceil + 1) + 4n^2 + 4m(\lceil \log_2 m \rceil + 1)(2h_0 + 1) + (2n + 12h_0 + 16)m$ bits. Therefore, it needed 11.9 KB (41.4 KB, resp.) for 32-bit (64-bit, resp.) integer division.

Though our protocol needs more data transfer, the round complexity becomes very small. In fact, in the WAN setting as described above, [12] needs 4968 ms (6264 ms, resp.) for 32-bit (64-bit, resp.) division only in latency, which is about 2 times slower than our protocol.

6. Future Work

In this section, we show some future work.

1. The necessity of bit expansion is a phenomenon not only in the division but also in logarithm, square root, and so on. Hence, there is the possibility to make these operations more efficient by applying MultBit protocol.
2. In this paper, we treated semi-honest security. Semi-honest secure protocols having certain properties can be easily extended to malicious security by using SPDZ [25] or SPDZ$_{2^k}$ [26]. However, it seems difficult to apply these methods to our protocols because our new building blocks constructed in Sect. 4 need some special operations on each shared value (such as local bit decomposition) and complicated correlated randomness.

Acknowledgements

This work was partly supported by JST CREST JPMJCR19F6, the Ministry of Internal Affairs and Communications SCOPE Grant Number 182103105, JST CREST JPMJCR14D6, and JSPS KAKENHI Grant Number JP21J20186.

[†]Also, [12] constructed an *exact* division protocol in the semi-honest model, which is the same setting as our protocol.

References

[1] K. Hiwatashi, S. Ohata, and K. Nuida, "An efficient secure division protocol using approximate multi-bit product and new constant-round building blocks," Applied Cryptography and Network Security - 18th International Conference, ACNS 2020, Lecture Notes in Computer Science, vol.12146, pp.357–376, Springer, 2020.
[2] A.C. Yao, "How to generate and exchange secrets (extended abstract)," 27th Annual Symposium on Foundations of Computer Science, pp.162–167, 1986.
[3] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-throughput semi-honest secure three-party computation with an honest majority," 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016, pp.805–817, ACM, 2016.
[4] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof, "Fast large-scale honest-majority MPC for malicious adversaries," Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Lecture Notes in Computer Science, vol.10993, pp.34–64, Springer, 2018.
[5] D. Demmler, T. Schneider, and M. Zohner, "ABY — A framework for efficient mixed-protocol secure two-party computation," 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, 2015.
[6] M. Ishaq, A.L. Milanova, and V. Zikas, "Efficient MPC via program analysis: A framework for efficient optimal mixing," 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, pp.1539–1556, ACM, 2019.
[7] F. Kerschbaum, T. Schneider, and A. Schröpfer, "Automatic protocol selection in secure two-party computations," Applied Cryptography and Network Security - 12th International Conference, ACNS 2014, Lecture Notes in Computer Science, vol.8479, pp.566–584, Springer, 2014.
[8] M. Aliasgari, M. Blanton, Y. Zhang, and A. Steele, "Secure computation on floating point numbers," 20th Annual Network and Distributed System Security Symposium, NDSS 2013, The Internet Society, 2013.
[9] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson, "High-performance secure multi-party computation for data mining applications," Int. J. Inf. Secur., vol.11, no.6, pp.403–418, 2012.
[10] O. Catrina and A. Saxena, "Secure computation with fixed-point numbers," Financial Cryptography and Data Security, 14th International Conference, FC 2010, Lecture Notes in Computer Science, vol.6052, pp.35–50, Springer, 2010.
[11] R. Lazzeretti and M. Barni, "Division between encrypted integers by means of garbled circuits," 2011 IEEE International Workshop on Information Forensics and Security, WIFS 2011, pp.1–6, IEEE Computer Society, 2011.
[12] H. Morita, N. Attrapadung, S. Ohata, K. Nuida, S. Yamada, K. Shimizu, G. Hanaoka, and K. Asai, "Secure division protocol and applications to privacy-preserving chi-squared tests," International Symposium on Information Theory and Its Applications, ISITA 2018, pp.530–534, IEEE, 2018.
[13] T. Veugen, "Encrypted integer division," 2010 IEEE International Workshop on Information Forensics and Security, WIFS 2010, pp.1–6, IEEE, 2010.
[14] T. Veugen, "Encrypted integer division and secure comparison," Int. J. Appl. Cryptogr., vol.3, no.2, pp.166–180, 2014.
[15] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," 2017 IEEE Symposium on Security and Privacy, S&P 2017, pp.19–38, 2017.
[16] I. Damgård, M. Fitzi, E. Kiltz, J.B. Nielsen, and T. Toft, "Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation," Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, Lecture Notes in Computer Science, vol.3876, pp.285–304, Springer, 2006.
[17] H. Morita, N. Attrapadung, T. Teruya, S. Ohata, K. Nuida, and

G. Hanaoka, "Constant-round client-aided secure comparison protocol," Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Lecture Notes in Computer Science, vol.11099, pp.395–415, Springer, 2018.

[18] T. Nishide and K. Ohta, "Multiparty computation for interval, equality, and comparison without bit-decomposition protocol," Public Key Cryptography - PKC 2007, 10th International Conference on Practice and Theory in Public-Key Cryptography, Lecture Notes in Computer Science, vol.4450, pp.343–360, Springer, 2007.

[19] R.E. Goldschmidt, "Applications of division by convergence," Ph.D. thesis, Massachusetts Institute of Technology, 1964.

[20] S. Ohata and K. Nuida, "Communication-efficient (client-aided) secure two-party protocols and its application," Financial Cryptography and Data Security - 24th International Conference, FC 2020, Lecture Notes in Computer Science, vol.12059, pp.369–385, Springer, 2020.

[21] O. Goldreich, The Foundations of Cryptography - Volume 2: Basic Applications, Cambridge University Press, 2004.

[22] S. Siim, "A comprehensive protocol suite for secure two-party computation," Master's Thesis, University of Tartu, 2016.

[23] M. Barni, J. Guajardo, and R. Lazzeretti, "Privacy preserving evaluation of signal quality with application to ECG analysis," 2010 IEEE International Workshop on Information Forensics and Security, WIFS 2010, pp.1–6, IEEE, 2010.

[24] M. Burkhart, M. Strasser, and X.A. Dimitropoulos, "SEPIA: Security through private information aggregation," CoRR, vol.abs/0903.4258, 2009.

[25] I. Damgård, V. Pastro, N.P. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Lecture Notes in Computer Science, vol.7417, pp.643–662, Springer, 2012.

[26] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing, "SPD$\mathbb{Z}$_2^k: Efficient MPC mod 2^k for dishonest majority," Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Lecture Notes in Computer Science, vol.10992, pp.769–798, Springer, 2018.



Keitaro Hiwatashi received the Master's degree at The University of Tokyo in 2021. He is currently a Doctoral course student in Graduate School of Information Science and Technology, The University of Tokyo. His research interest is mainly in cryptography and information security.



Satsuya Ohata received B.Eng. degree at Chiba University in 2011 and Ph.D. (Information Science and Technology) degree at The University of Tokyo in 2016. He is currently a senior researcher at Digital Garage, Inc. His research interests are cryptography and information security.



Koji Nuida received the Ph.D. degree in Mathematical Science from The University of Tokyo, Japan, in 2006. Currently, he is mainly working as a professor at Institute of Mathematics for Industry (IMI), Kyushu University, Japan. His research interest is mainly in mathematics and mathematical cryptography.