

Communication-Efficient (Client-Aided) Secure Two-Party Protocols and Its Application

Satsuya Ohata¹ and Koji Nuida^{1,2}

¹ National Institute of Advanced Industrial Science and Technology, Tokyo, Japan
satsuya.ohata@aist.go.jp

² The University of Tokyo, Tokyo, Japan nuida@mist.i.u-tokyo.ac.jp

Abstract. Secure multi-party computation (MPC) allows a set of parties to compute a function jointly while keeping their inputs private. Compared with the MPC based on garbled circuits, some recent research results show that MPC based on secret sharing (SS) works at a very high speed. Moreover, SS-based MPC can be easily vectorized and achieve higher throughput. In SS-based MPC, however, we need many communication rounds for computing concrete protocols like equality check, less-than comparison, etc. This property is not suited for large-latency environments like the Internet (or WAN). In this paper, we construct semi-honest secure communication-efficient two-party protocols. The core technique is *Beaver triple extension*, which is a new tool for treating multi-fan-in gates, and we also show how to use it efficiently. We mainly focus on reducing the number of communication rounds, and our protocols also succeed in reducing the number of communication bits (in most cases). As an example, we propose a less-than comparison protocol (under practical parameters) with *three* communication rounds. Moreover, the number of communication bits is also 38.4% fewer. As a result, total online execution time is 56.1% shorter than the previous work adopting the same settings. Although the computation costs of our protocols are more expensive than those of previous work, we confirm via experiments that such a disadvantage has small effects on the whole online performance in the typical WAN environments.

1 Introduction

Secure multi-party computation (MPC) [33,17] allows a set of parties to compute a function f jointly while keeping their inputs private. More precisely, the $N(\geq 2)$ parties, each holding private input x_i for $i \in [1, N]$, are able to compute the output $f(x_1, \dots, x_N)$ without revealing their private inputs x_i . Some recent research showed there are many progresses in the research on MPC based on secret sharing (SS) and its performance is dramatically improved. SS-based MPC can be easily vectorized and suitable for parallel executions. We can obtain large throughput in SS-based MPC since we have no limit on the size of vectors. This is a unique property on SS-based MPC, and it is compatible with the SIMD operations like mini-batch training in privacy-preserving machine learning. We cannot enjoy this advantage in the MPC based on garbled circuits (GC)

or homomorphic encryption (HE). The most efficient MPC scheme so far is three-party computation (3PC) based on 2-out-of-3 SS (e.g., [2,9]). In two-party computation (2PC), which is the focus of this paper, we need fewer hardware resources than 3PC. Although it does not work at high speed since we need heavy pre-computation, we can mitigate this problem by adopting slightly new MPC models like client/server-aided models that we denote later.

In addition to the advantage as denoted above, the amount of data transfer in online phase is also small in SS-based MPC than GC/HE-based one. However, the number of communication rounds we need for computation is large in SS-based MPC. We need one interaction between computing parties when we compute an arithmetic multiplication gate or a boolean AND gate, which is time-consuming when processing non-linear functions since it is difficult to make the circuit depth shallow. This is a critical disadvantage in real-world privacy-preserving applications since there are non-linear functions we frequently use in practice like equality check, less-than comparison, max value extraction, activation functions in machine learning, etc. In most of the previous research, however, this problem has not been seriously treated. This is because they assumed there is (high-speed) LAN connection between computing parties. Under such environments, total online execution time we need for processing non-linear functions is small even if we need many interactions between computing parties since the communication latency is usually very short (typically $\leq 0.5\text{ms}$). This assumption is somewhat strange in practice, as the use of LAN suggests that MPC is executed on the network that is maintained by the same administrator/organization. In that case, it is not clear if the requirement for SS that parties do not collude is held or not. Hence, it looks more suitable to assume non-local networks like WAN. However, large communication latency in WAN becomes the performance bottleneck in SS-based MPC. We find by our experiments that the time caused by the communication latency occupies more than 99% in some cases for online total execution time. To reduce the effect of the large communication latency, it is important to develop SS-based MPC with fewer communication rounds. In other words, we should put in work to make the circuit shallower to improve the concrete efficiency of SS-based MPC.

1.1 Related Work

MPC Based on Secret Sharing There are many research results on SS-based MPC. For example, we have results on highly-efficient MPC (e.g., [2,9]), concrete tools or the toolkit (e.g., [12,28,5,27]), mixed-protocol framework [13,31,24], application to privacy-preserving machine learning or data analysis (e.g., [26,21,31,24,10]), proposal of another model for speeding up the pre-computation [23,26], etc. As denoted previously, however, we have not been able to obtain good experimental results for computing large circuits over WAN environments. For example, [26] denoted the neural network training on WAN setting is not practical yet.

MPC Based on Garbled Circuit or Homomorphic Encryption There are also many research results on GC/HE-based MPC. For example, we have results

on the toolkit (e.g., [22]), encryption switching protocols [20,11], application to private set intersection (e.g., [30]) or privacy-preserving machine learning (e.g., [6,15,29,7,18,19]), etc. Recently, we have many research results on GC for more than three parties (e.g., [25,35]) and Arithmetic GC (e.g., [1]). Note that it is difficult to improve the circuit size on standard boolean GC [34], which is a bottleneck on GC-based MPC. Moreover, [4,8] proposed the GC-based MPC for WAN environments and showed the benchmark using AES, etc. Even if we adopt the most efficient GC [34] with 128-bit security, however, we need to send at least 256-bit string per an AND gate. This is two orders of magnitude larger than SS-based MPC. We construct the round-efficient protocol while keeping data traffic small.

1.2 Our Contribution

There are two main contributions in this paper. First, we propose the method for treating multi-fan-in gates in semi-honest secure SS-based 2PC and show how to use them efficiently. Second, we propose many round-efficient protocols and show their performance evaluations via experiments. We explain the details of them as follows:

1. We propose the method for treating multi-fan-in MULT/AND gates over \mathbb{Z}_{2^n} and some techniques for reducing the communication rounds of protocols. Our N -fan-in gates are based on the extension of Beaver triples, which is a technique for computing standard 2-fan-in gates. In our technique, however, we have a disadvantage that the computation costs and the memory costs are exponentially increased by N ; that is, we have to limit the size of N in practice. On the other hand, we can improve the costs of communication. More concretely, we can compute arbitrary N -fan-in MULT/AND with one communication round and the amount of data transfer is also improved. Moreover, we show performance evaluation results on above multi-fan-in gates via experiments. More concretely, see Sections 3 and 5.1.
2. We propose round-efficient protocols using multi-fan-in gates. We need fewer interactions for our protocols between computing parties in online phase than previous ones. When we use shares over $\mathbb{Z}_{2^{32}}$, compared with the previous work [5], we reduce the communication rounds as follows: **Equality** : (5 \rightarrow 2), **Comparison** : (7 \rightarrow 3), and **Max** for 3 elements:(18 \rightarrow 4). Moreover, we show the performance evaluation results on our protocols via experiments. From our experiments, we find the computation costs for multi-fan-in gates and protocols based on them have small effects on the whole online performance in the typical WAN environments. We also implement an application (a privacy-preserving exact edit distance protocol for genome strings) using our protocols. More concretely, see Sections 4, 5.2, and 5.3.

2 Preliminaries

2.1 Syntax for Secret Sharing

A 2-out-of-2 secret sharing ((2, 2)-SS) scheme over \mathbb{Z}_{2^n} consists of two algorithms: Share and Reconst. Share takes as input $x \in \mathbb{Z}_{2^n}$, and outputs $(\llbracket x \rrbracket_0, \llbracket x \rrbracket_1) \in \mathbb{Z}_{2^n}^2$, where the bracket notation $\llbracket x \rrbracket_i$ denotes the share of the i -th party (for $i \in \{0, 1\}$). We denote $\llbracket x \rrbracket = (\llbracket x \rrbracket_0, \llbracket x \rrbracket_1)$ as their shorthand. Reconst takes as input $\llbracket x \rrbracket$, and outputs x . For arithmetic sharing $\llbracket x \rrbracket^A = (\llbracket x \rrbracket_0^A, \llbracket x \rrbracket_1^A)$ and boolean sharing $\llbracket x \rrbracket^B = (\llbracket x \rrbracket_0^B, \llbracket x \rrbracket_1^B)$, we consider power-of-two integers n (e.g. $n = 64$) and $n = 1$, respectively.

2.2 Secure Two-Party Computation Based on (2,2)-Additive Secret Sharing

Here, we explain how to compute arithmetic ADD/MULT gates on (2, 2)-additive SS. We use the standard (2, 2)-additive SS scheme, defined by

- Share(x): randomly choose $r \in \mathbb{Z}_{2^n}$ and let $\llbracket x \rrbracket_0^A = r$ and $\llbracket x \rrbracket_1^A = x - r \in \mathbb{Z}_{2^n}$.
- Reconst($\llbracket x \rrbracket_0^A, \llbracket x \rrbracket_1^A$): output $\llbracket x \rrbracket_0^A + \llbracket x \rrbracket_1^A$.

We can compute fundamental operations; that is, $\text{ADD}(x, y) := x + y$ and $\text{MULT}(x, y) := xy$. $\llbracket z \rrbracket \leftarrow \text{ADD}(\llbracket x \rrbracket, \llbracket y \rrbracket)$ can be done locally by just adding each party's shares on x and on y . $\llbracket w \rrbracket \leftarrow \text{MULT}(\llbracket x \rrbracket, \llbracket y \rrbracket)$ can be done in various ways. We will use the standard method based on Beaver triples (BT) [3]. Such a triple consists of $\text{bt}_0 = (a_0, b_0, c_0)$ and $\text{bt}_1 = (a_1, b_1, c_1)$ such that $(a_0 + a_1)(b_0 + b_1) = (c_0 + c_1)$. Hereafter, a , b , and c denote $a_0 + a_1$, $b_0 + b_1$, and $c_0 + c_1$, respectively. We can compute these BT in offline phase. In this protocol, each i -th party P_i ($i \in \{0, 1\}$) can compute the multiplication share $\llbracket z \rrbracket_i = \llbracket xy \rrbracket_i$ as follows: (1) P_i first compute $(\llbracket x \rrbracket_i - a_i)$ and $(\llbracket y \rrbracket_i - b_i)$. (2) P_i sends them to P_{1-i} . (3) P_i reconstruct $x' = x - a$ and $y' = y - b$. (4) P_0 computes $\llbracket z \rrbracket_0 = x'y' + x'b_0 + y'a_0 + c_0$ and P_1 computes $\llbracket z \rrbracket_1 = x'b_1 + y'a_1 + c_1$. Here, $\llbracket z \rrbracket_0$ and $\llbracket z \rrbracket_1$ calculated as above procedures are valid shares of xy ; that is, $\text{Reconst}(\llbracket z \rrbracket_0, \llbracket z \rrbracket_1) = xy$. We abuse notations and write the ADD and MULT protocols simply as $\llbracket x \rrbracket + \llbracket y \rrbracket$ and $\llbracket x \rrbracket \cdot \llbracket y \rrbracket$, respectively. Note that similarly to the ADD protocol, we can also locally compute multiplication by constant c , denoted by $c \cdot \llbracket x \rrbracket$.

We can easily extend above protocols to boolean gates. By converting $+$ and $-$ to \oplus in arithmetic ADD and MULT protocols, we can obtain XOR and AND protocols, respectively. We can construct NOT and OR protocols from the properties of these gates. When we compute $\text{NOT}(\llbracket x \rrbracket_0^B, \llbracket x \rrbracket_1^B)$, P_0 and P_1 output $\neg \llbracket x \rrbracket_0^B$ and $\llbracket x \rrbracket_1^B$, respectively. When we compute $\text{OR}(\llbracket x \rrbracket, \llbracket y \rrbracket)$, we compute $\neg \text{AND}(\neg \llbracket x \rrbracket, \neg \llbracket y \rrbracket)$. We abuse notations and write the XOR, AND, NOT, and OR protocols simply as $\llbracket x \rrbracket \oplus \llbracket y \rrbracket$, $\llbracket x \rrbracket \wedge \llbracket y \rrbracket$, $\neg \llbracket x \rrbracket$ (or $\overline{\llbracket x \rrbracket}$), and $\llbracket x \rrbracket \vee \llbracket y \rrbracket$, respectively.

2.3 Semi-Honest Security and Client-Aided Model

In this paper, we consider simulation-based security notion in the presence of semi-honest adversaries (for 2PC) as in [16]. We show the definition in Appendix A. As described in [16], composition theorem for the semi-honest model

holds; that is, any protocol is privately computed as long as its subroutines are privately computed.

In this paper, we adopt client-aided model [26,27] (or server-aided model [23]) for 2PC. In this model, a client (other than computing parties) generates and distributes shares of secrets. Moreover, the client also generates and distributes some necessary BTs to the computing parties. This improves the efficiency of offline computation dramatically since otherwise computing parties would have to generate BTs by themselves jointly via heavy cryptographic primitives like homomorphic encryption or oblivious transfer. The only downside for this model is the restriction that any computing party is assumed to not collude with the client who generates BTs for keeping the security.

3 Core Tools for Round-Efficient Protocols

In this section, we propose a core tool for round-efficient 2PC that we call “Beaver triple extension (BTE)”. Moreover, we explain some techniques for pre-computation to reduce the communication rounds in online phase.

3.1 Example: 3-fan-in MULT/AND via 3-Beaver Triple Extension

Here, we explain the case of 3-fan-in gates as an example. We consider how to extend the mechanism of a 2-fan-in MULT gate to a 3-fan-in MULT gate (3-MULT); that is, we consider how to construct a special BT that cancels the terms coming out from $(x - a)(y - b)(z - c)$ other than xyz . We can obtain such one by extending the standard BT. It consists of $(a_0, b_0, c_0, d_0, e_0, f_0, g_0)$ for P_0 and $(a_1, b_1, c_1, d_1, e_1, f_1, g_1)$ for P_1 satisfying the conditions $a_0 + a_1 = a, \dots, g_0 + g_1 = g, ab = d, bc = e, ca = f$, and $abc = g$. We call the above special BT as 3-Beaver triple extension (3-BTE) in this paper. We can compute the 3-MULT using above 3-BTE as follows:

1. P_i ($i \in \{0, 1\}$) compute $(\llbracket x \rrbracket_i - a_i)$, $(\llbracket y \rrbracket_i - b_i)$, and $(\llbracket z \rrbracket_i - c_i)$.
2. P_i send them to another party.
3. P_i reconstruct $x' = x - a$, $y' = y - b$, and $z' = z - c$.
4. P_0 computes $\llbracket w \rrbracket_0 = x'y'z' + x'y'c_0 + y'z'a_0 + z'x'b_0 + x'e_0 + y'f_0 + z'd_0 + g_0$
and P_1 computes $\llbracket w \rrbracket_1 = x'y'c_1 + y'z'a_1 + z'x'b_1 + x'e_1 + y'f_1 + z'd_1 + g_1$.

$\llbracket w \rrbracket_0$ and $\llbracket w \rrbracket_1$ are valid shares of xyz . We can obviously construct a boolean 3-BTE and 3-fan-in AND gate (3-AND) by converting $+$ and $-$ to \oplus in the 3-MULT case and also obtain 3-fan-in OR gates (3-OR).

3.2 N -fan-in MULT/AND via N -Beaver Triple Extension

N -Beaver Triple Extension Let N be a positive integer. Let $\mathcal{M} = \mathbb{Z}_M$ for some M (e.g., $M = 2^n$). Write $[1, N] = \{1, 2, \dots, N\}$. We define a client-aided protocol for generating N -BTE as follows:

1. Client randomly chooses $\llbracket a_{\{\ell\}} \rrbracket_0$ and $\llbracket a_{\{\ell\}} \rrbracket_1$ from \mathcal{M} ($\ell = 1, \dots, N$). Let $a_{\{\ell\}} \leftarrow \llbracket a_{\{\ell\}} \rrbracket_0 + \llbracket a_{\{\ell\}} \rrbracket_1$. For each $I \subseteq [1, N]$ with $|I| \geq 2$, by setting $a_I \leftarrow \prod_{\ell \in I} a_{\{\ell\}}$, client randomly chooses $\llbracket a_I \rrbracket_0 \in \mathcal{M}$ and sets $\llbracket a_I \rrbracket_1 \leftarrow a_I - \llbracket a_I \rrbracket_0$.
2. Client sends all the $\llbracket a_I \rrbracket_0$ to P_0 and all the $\llbracket a_I \rrbracket_1$ to P_1 .

Note that, in the protocol above, the process of randomly choosing $\llbracket a_I \rrbracket_0$ and then setting $\llbracket a_I \rrbracket_1 \leftarrow a_I - \llbracket a_I \rrbracket_0$ is equivalent to randomly choosing $\llbracket a_I \rrbracket_1$ and then setting $\llbracket a_I \rrbracket_0 \leftarrow a_I - \llbracket a_I \rrbracket_1$. Therefore, the roles of P_0 and P_1 are symmetric.

Multiplication Protocol For $\ell = 1, \dots, N$, let $(\llbracket x_\ell \rrbracket_0, \llbracket x_\ell \rrbracket_1)$ be given shares of ℓ -th secret input value $x_\ell \in \mathcal{M}$. The protocol for multiplication is constructed as follows:

1. Client generates and distributes N -BTE $(\llbracket a_I \rrbracket_0)_I$ and $(\llbracket a_I \rrbracket_1)_I$ to the two parties as described above.
2. For $k = 0, 1$, P_k computes $\llbracket x'_\ell \rrbracket_k \leftarrow \llbracket x_\ell \rrbracket_k - \llbracket a_{\{\ell\}} \rrbracket_k$ for $\ell = 1, \dots, N$ and sends those $\llbracket x'_\ell \rrbracket_k$ to P_{1-k} .
3. For $k = 0, 1$, P_k computes $x'_\ell \leftarrow \llbracket x'_\ell \rrbracket_{1-k} + \llbracket x'_\ell \rrbracket_k$ for $\ell = 1, \dots, N$.
4. P_0 outputs $\llbracket y \rrbracket_0$ given by

$$\llbracket y \rrbracket_0 \leftarrow \prod_{\ell=1}^N x'_\ell + \sum_{\emptyset \neq I \subseteq [1, N]} \left(\prod_{\ell \in [1, N] \setminus I} x'_\ell \right) \llbracket a_I \rrbracket_0$$

while P_1 outputs $\llbracket y \rrbracket_1$ given by

$$\llbracket y \rrbracket_1 \leftarrow \sum_{\emptyset \neq I \subseteq [1, N]} \left(\prod_{\ell \in [1, N] \setminus I} x'_\ell \right) \llbracket a_I \rrbracket_1 .$$

We can prove the correctness and semi-honest security of this protocol. Due to the page limitation, we show the proofs in Appendix B.

3.3 Discussion on Beaver Triple Extension

We can achieve the same functionality of N -MULT/AND by using 2-MULT/AND multiple times and there are some trade-offs between these two strategies.

Memory, Computation, and Communication Costs In the computation of N -fan-in MULT/AND using N -BTE, the memory consumption and computation cost increase exponentially with N . Therefore, we have to put a restriction on the size of N and concrete settings change optimal N . In this paper, we use N -MULT/AND for $N \leq 9$ to construct round-efficient protocols.

N -fan-in MULT/AND using N -BTE needs fewer communication costs. Notably, the number of communication rounds of our protocol does not depend on N and this improvement has significant effects on practical performances in

WAN settings. Because of the problems on the memory/computation costs we denoted above, however, there is a limitation for the size of N . When we use L -fan-in MULT/AND ($L \leq N$) gates, we need $\frac{\lceil \log N \rceil}{\lceil \log L \rceil}$ communication rounds for computing N -fan-in MULT/AND. When we set $L = 8$, for example, we need two communication rounds to compute a 64-fan-in AND.

Comparison with Previous Work Damgård et al. [12] also proposed how to compute N -fan-in gates in a round-efficient manner using Lagrange interpolation. Each of their scheme and ours has its merits and demerits. Their scheme has an advantage over memory consumption and computational costs; that is, their N -fan-in gates do not need exponentially large memory and computation costs. On the other hand, their scheme needs two communication rounds to compute N -fan-in gates for any N and requires the share spaces to be \mathbb{Z}_p (p : prime). A 2PC scheme over \mathbb{Z}_{2^n} is sometimes more efficient than one over \mathbb{Z}_p when we implement them using low-level language (e.g., C++) since we do not have to compute remainders modulo 2^n for all arithmetic operations.

3.4 More Techniques for Reducing Communication Rounds

On Weights At Most One We consider the plain input x that all bits are 0, or only a single bit is 1 and others are 0. For example, we consider $x = 00100000$ and its boolean shares $(\llbracket x \rrbracket_0^{\mathbb{B}}, \llbracket x \rrbracket_1^{\mathbb{B}}) = (10011011, 10111011)$. We find these are correct boolean shares of x since $\llbracket x \rrbracket_0^{\mathbb{B}} \oplus \llbracket x \rrbracket_1^{\mathbb{B}} = x$ holds. In this setting, we can compute the share representing whether all the bits of x are 0 or not without communications between P_0 and P_1 . More concretely, we can compute it by locally computing XOR for all bits on each share. In the above example, P_0 and P_1 compute $\bigoplus \llbracket x \rrbracket_0^{\mathbb{B}} = 1$ and $\bigoplus \llbracket x \rrbracket_1^{\mathbb{B}} = 0$, respectively. $1 \oplus 0 = 1$ means there is 1 in x . This technique is implicitly used in the previous work [5] for constructing an arithmetic overflow detection protocol (**Overflow**), which is an important building block for constructing less-than comparison and more. We show more skillful use of this technique for constructing **Overflow** to avoid heavy computation in our protocols. More concretely, see Section 4.2.

Arithmetic Blinding We consider the situation that two clients who have secrets also execute computation (i.e., an input party is equal to a computing party), which is the different setting from client-aided 2PC. In this case, P_0 and P_1 randomly split the secret x and y into x_0, x_1 and y_0, y_1 , respectively. Then P_0 sends x_1 to P_1 and P_1 sends y_0 to P_0 . If P_0 and P_1 previously obtain a_0, b_0, c_0 and a_1, b_1, c_1 , respectively, P_0 and P_1 can compute $\llbracket z \rrbracket = xy$ via the standard multiplication protocol. During this procedure, both P_0 and P_1 obtain $x - a$ and $y - b$. Here, P_0 finds a and P_1 finds b since P_0 and P_1 know the value of x and y , respectively. Therefore, it does not matter if P_0 and P_1 previously know the corresponding values; that is, P_0 can send b_0 to P_1 and P_1 can send a_1 to P_0 in the pre-computation phase. This operation does not cause security problems.

By above pre-processing, P_0 and P_1 can directly send $x - a$ and $y - b$ in the multiplication protocol, respectively. As a result, we can reduce the amount of data transfer in the multiplication protocol. Note that in the setting that the input party is not equal to the computing party (e.g., standard client-aided 2PC), this pre-processing does not work well since P_0 and P_1 do not have x and y , respectively and cannot compute $\llbracket z \rrbracket = xy$ correctly. Even in the client-aided 2PC setting, however, this situation appears in the boolean-to-arithmetic conversion protocol. More concretely, see Section 4.3.

Trivial Sharing We consider the setting that an input party is not equal to a computing party, which is the same one as standard client-aided 2PC. In this situation, we can use the share $\llbracket b \rrbracket_i$ ($i \in \{0, 1\}$) itself as a secret value for computations by considering another party has the share $\llbracket 0 \rrbracket_{1-i}$. Although we find this technique in the previous work [5], we can further reduce the communication rounds of two-party protocols by combining this technique and BTE. More concretely, see Section 4.3.

4 Communication-Efficient Protocols

In this section, we show round-efficient 2PC protocols using BTE and the techniques in Section 3.4. For simplicity, in this section, we set a share space to $\mathbb{Z}_{2^{16}}$ and use N -fan-in gates ($N \leq 5$) to explain our proposed protocols. Although we omit the protocols over $\mathbb{Z}_{2^{32}}/\mathbb{Z}_{2^{64}}$ due to the page limitation, we can obtain the protocols with the same communication rounds with $\mathbb{Z}_{2^{16}}$ by using 7 or less fan-in AND over $\mathbb{Z}_{2^{32}}$ and 9 or less fan-in AND over $\mathbb{Z}_{2^{64}}$. We omit the correctness of the protocols adopting the same strategy in the previous work [5].

4.1 Equality Check Protocol and Its Application

An equality check protocol $\text{Equality}(\llbracket x \rrbracket^A, \llbracket y \rrbracket^A)$ outputs $\llbracket z \rrbracket^B$, where $z = 1$ iff $x = y$. We start from the approach by [5] and focus on reducing communication rounds. In Equality , roughly speaking, we first compute $t = x - y$ and then check if all bits of t are 0 or not. If all the bits of t are 0, it means $t = x - y = 0$. Although we can perform this functionality via 16-OR, we cannot directly execute such a large-fan-in OR gate. We need $\log_2 16 = 4$ communication rounds for the above procedure if we only use 2-OR with a tree structure. However, if we can use 4-OR, we can execute Equality with $\log_4 16 = 2$ communication rounds. We show our two-round Equality as in Algorithm 1: In this strategy, more generally, we need $\frac{\lceil \log n \rceil}{\lceil \log L \rceil}$ communication rounds for executing Equality when we set the share space to \mathbb{Z}_{2^n} and use N -OR ($N \leq L$).

We can also obtain a round-efficient table lookup protocol TLU (or, 1-out-of- L oblivious transfer) using our Equality . We show the construction of three-round TLU in Appendix C.1.

Algorithm 1 Our Proposed Equality**Functionality:** $\llbracket z \rrbracket^{\text{B}} \leftarrow \text{Equality}(\llbracket x \rrbracket^{\text{A}}, \llbracket y \rrbracket^{\text{A}})$ **Ensure:** $\llbracket z \rrbracket^{\text{B}}$, where $z = 1$ iff $x = y$.

- 1: P_0 and P_1 locally compute $\llbracket t \rrbracket_0^{\text{A}} = \llbracket x \rrbracket_0^{\text{A}} - \llbracket y \rrbracket_0^{\text{A}}$ and $\llbracket t \rrbracket_1^{\text{A}} = \llbracket y \rrbracket_1^{\text{A}} - \llbracket x \rrbracket_1^{\text{A}}$, respectively.
- 2: P_i ($i \in \{0, 1\}$) locally extend $\llbracket t \rrbracket_i^{\text{A}}$ to binary and see them as boolean shares; that is, P_i obtain $\llbracket t[15] \rrbracket_i^{\text{B}}, \dots, \llbracket t[0] \rrbracket_i^{\text{B}}$.
- 3: P_i compute $\llbracket t'[j] \rrbracket^{\text{B}} \leftarrow 4\text{-OR}(\llbracket t[4j] \rrbracket^{\text{B}}, \llbracket t[4j+1] \rrbracket^{\text{B}}, \llbracket t[4j+2] \rrbracket^{\text{B}}, \llbracket t[4j+3] \rrbracket^{\text{B}})$ for $j \in [0, \dots, 3]$.
- 4: P_i compute $\llbracket t'' \rrbracket^{\text{B}} \leftarrow 4\text{-OR}(\llbracket t'[0] \rrbracket^{\text{B}}, \llbracket t'[1] \rrbracket^{\text{B}}, \llbracket t'[2] \rrbracket^{\text{B}}, \llbracket t'[3] \rrbracket^{\text{B}})$.
- 5: P_i compute $\llbracket z \rrbracket^{\text{B}} = \neg \llbracket t'' \rrbracket^{\text{B}}$.
- 6: **return** $\llbracket z \rrbracket^{\text{B}}$.

4.2 Overflow Detection Protocol and Applications

An arithmetic overflow detection protocol **Overflow** has many applications and is also a core building block of less-than comparison protocol. The same as the approach by [5], we construct **Overflow** via the most significant non-zero bit extraction protocol **MSNZB**. We first explain how to construct **MSNZB** efficiently and then show two-round **Overflow**.

A protocol for extracting the most significant non-zero bit ($\text{MSNZB}(\llbracket x \rrbracket^{\text{B}} = \llbracket x[15] \rrbracket^{\text{B}}, \dots, \llbracket x[0] \rrbracket^{\text{B}})$) finds the position of the first “1” of the x and outputs such a boolean share vector $\llbracket z \rrbracket^{\text{B}} = \llbracket z[15] \rrbracket^{\text{B}}, \dots, \llbracket z[0] \rrbracket^{\text{B}}$; that is, for example, if $x = 0010011100010000$, then $z = 0010000000000000$. To find the position of the first “1” in x in a privacy-preserving manner, we use a “prefix-OR” operation [5]. In this procedure, we first replace further to the right bits than leftmost 1 with 1 via 2-OR gates and obtain $x' = 0 \dots 011 \dots 1$. Then, we compute $z = x' \oplus (x' \gg 1)$. In this **MSNZB**, we need four communication rounds since 2-OR runs four times even if we parallelize the processing. Intuitively, we can construct two-round **MSNZB** via 4-OR; that is, we compute multi-fan-in prefix-OR using N -OR ($N \leq 4$). In this intuitive two-round **MSNZB**, however, computation costs significantly increase since we have to compute 4-OR many times. Therefore, we consider how to reduce them while keeping the number of communication rounds. We show our two-round **MSNZB** as in Algorithm 2. In this construction, we first separate a bit string into some blocks and compute in-block **MSNZB**. Then, we compute correct **MSNZB** for x via in-block **MSNZB**. In Algorithm 2, we separate 16-bit string uniformly into 4 blocks for avoiding the usage of large fan-in OR. This **MSNZB** is more efficient than the intuitive construction since we use fewer ($= 4 + 4$) 4-fan-in gates.

Based on the above **MSNZB**, we can construct an arithmetic overflow detection protocol **Overflow**($\llbracket x \rrbracket^{\text{A}}, k$). This protocol outputs $\llbracket z \rrbracket^{\text{B}}$, where $z = 1$ iff the condition $(\llbracket x \rrbracket_0^{\text{A}} \bmod 2^k + \llbracket x \rrbracket_1^{\text{A}} \bmod 2^k) \geq 2^k$ holds. **Overflow** is an important building block of many other protocols that appear in the later of this section. We also start from the approach by [5]. In their **Overflow**, we check whether or not there exists 1 in $u = (-\llbracket x \rrbracket_1 \bmod 2^k)$ at the same position of **MSNZB** on $d = ((\llbracket x \rrbracket_0 \bmod 2^k) \oplus (-\llbracket x \rrbracket_1 \bmod 2^k))$. Even if we apply our two-round

Algorithm 2 Our Proposed MSNZB**Functionality:** $\llbracket z \rrbracket^B \leftarrow \text{MSNZB}(\llbracket x \rrbracket^B)$ **Ensure:** $\llbracket z \rrbracket^B = [\llbracket z[15] \rrbracket^B, \dots, \llbracket z[0] \rrbracket^B]$, where $z[j] = 1$ for the largest value j such that $x[j] = 1$ and $z[k] = 0$ for all $j \neq k$.

- 1: P_i ($i \in \{0, 1\}$) set $\llbracket t[j] \rrbracket_i^B = \llbracket x[j] \rrbracket_i^B$ for $j \in [3, 7, 11, 15]$. Then P_i parallelly compute $\llbracket t[j] \rrbracket_i^B \leftarrow 2\text{-OR}(\llbracket x[j] \rrbracket_i^B, \llbracket x[j+1] \rrbracket_i^B)$ for $j \in [2, 6, 10, 14]$, $\llbracket t[j] \rrbracket_i^B \leftarrow 3\text{-OR}(\llbracket x[j] \rrbracket_i^B, \llbracket x[j+1] \rrbracket_i^B, \llbracket x[j+2] \rrbracket_i^B)$ for $j \in [1, 5, 9, 13]$, and $\llbracket t[j] \rrbracket_i^B \leftarrow 4\text{-OR}(\llbracket x[j] \rrbracket_i^B, \llbracket x[j+1] \rrbracket_i^B, \llbracket x[j+2] \rrbracket_i^B, \llbracket x[j+3] \rrbracket_i^B)$ for $j \in [0, 4, 8, 12]$.
- 2: P_i compute $\llbracket t'[j] \rrbracket_i^B = \llbracket t[j] \rrbracket_i^B$ for $j \in [3, 7, 11, 15]$ and compute $\llbracket t'[j] \rrbracket_i^B = \llbracket t[j] \rrbracket_i^B \oplus \llbracket t[j+1] \rrbracket_i^B$ for $j \in [0, 1, 2, 4, 5, 6, 8, 9, 10, 12, 13, 14]$.
- 3: P_i locally compute $\llbracket s[j] \rrbracket_i^B = \bigoplus_{k=4j}^{4j+3} \llbracket t'[k] \rrbracket_i^B$ for $j \in [1, 2, 3]$.
- 4: P_i compute $\llbracket z[j] \rrbracket_i^B = \llbracket t'[j] \rrbracket_i^B$ for $j \in [12, \dots, 15]$. Then P_i parallelly compute $\llbracket z[j] \rrbracket_i^B \leftarrow 2\text{-AND}(\llbracket t'[j] \rrbracket_i^B, \neg \llbracket s[3] \rrbracket_i^B)$ for $j \in [8, \dots, 11]$, $\llbracket z[j] \rrbracket_i^B \leftarrow 3\text{-AND}(\llbracket t'[j] \rrbracket_i^B, \neg \llbracket s[2] \rrbracket_i^B, \neg \llbracket s[3] \rrbracket_i^B)$ for $j \in [4, \dots, 7]$, and $\llbracket z[j] \rrbracket_i^B \leftarrow 4\text{-AND}(\llbracket t'[j] \rrbracket_i^B, \neg \llbracket s[1] \rrbracket_i^B, \neg \llbracket s[2] \rrbracket_i^B, \neg \llbracket s[3] \rrbracket_i^B)$ for $j \in [0, \dots, 3]$.
- 5: **return** $\llbracket z \rrbracket^B = [\llbracket z[15] \rrbracket^B, \dots, \llbracket z[0] \rrbracket^B]$.

MSNZB in this section, we need three communication rounds for their Overflow since we need one more round to check the above condition using 2-AND. Here, we consider further improvements by combining MSNZB and 2-AND; that is, we increase the fan-in of AND on the step 4 in Algorithm 2 and push the computation of 2-AND into that step as in Algorithm 3: In our Overflow, we need a communication for the steps 2 and 6 in Algorithm 3 and succeed in constructing two-round Overflow using N -AND ($N \leq 5$) over $\mathbb{Z}_{2^{16}}$. If we set the share space to $\mathbb{Z}_{2^{32}}/\mathbb{Z}_{2^{64}}$, we need to use N -AND for $N \leq 7/N \leq 9$ for constructing two-round Overflow, respectively. Moreover, in Appendix D, we show more round-efficient Overflow. Although we need more computation and data transfer than Overflow in this section, we can compute Overflow with one communication round (for small share spaces in practice).

We have many applications of Overflow. We show the concrete construction of less-than comparison (Comparison) in Appendix C.2, which is a building block of the maximum value extraction protocol. In particular, thanks to the round-efficient Overflow, we can obtain a three-round Comparison. Morita et al. [27] proposed a constant (= five)-round Comparison using multi-fan-in gates that works under the shares over \mathbb{Z}_p [12]. Our Comparison is more round-efficient than theirs under the parameters we consider in this paper.

4.3 Boolean-to-Arithmetic Conversion Protocol and Extensions

A boolean-to-arithmetic conversion protocol B2A($\llbracket x \rrbracket^B$) outputs $\llbracket z \rrbracket^A$, where $z = x$. In (1-bit) boolean shares, there are four cases; that is, $(\llbracket x \rrbracket_0^B, \llbracket x \rrbracket_1^B) = (0, 0), (0, 1), (1, 0), (1, 1)$. Even if we consider these boolean shares as arithmetic ones, it works well in the first three cases; that is, $0 \oplus 0 = 0 + 0$, $0 \oplus 1 = 0 + 1$, and $1 \oplus 0 = 1 + 0$. However, $1 \oplus 1 \neq 1 + 1$ and we have to correct the output of this case. Based on this idea and the technique in Section 3.4 (trivial sharing), [5] pro-

Algorithm 3 Our Proposed Overflow

Functionality: $\llbracket z \rrbracket^B \leftarrow \text{Overflow}(\llbracket x \rrbracket^A, k)$
Ensure: $\llbracket z \rrbracket^B$, where $z = 1$ iff $(\llbracket x \rrbracket_0^A \bmod 2^k) + (\llbracket x \rrbracket_1^A \bmod 2^k) \geq 2^k$.

- 1: P_0 locally extends $(\llbracket x \rrbracket_0^A \bmod 2^k)$ to binary and obtains $\llbracket d \rrbracket_0^B = [\llbracket d[15] \rrbracket_0^B, \dots, \llbracket d[0] \rrbracket_0^B]$. P_1 also locally extends $(-\llbracket x \rrbracket_1^A \bmod 2^k)$ to binary and obtains $\llbracket d \rrbracket_1^B = [\llbracket d[15] \rrbracket_1^B, \dots, \llbracket d[0] \rrbracket_1^B]$.
 - 2: P_i ($i \in \{0, 1\}$) set $\llbracket t[j] \rrbracket_i^B = \llbracket d[j] \rrbracket_i^B$ for $j \in [3, 7, 11, 15]$. Then P_i parallelly compute $\llbracket t[j] \rrbracket_i^B \leftarrow 2\text{-OR}(\llbracket d[j] \rrbracket_i^B, \llbracket d[j+1] \rrbracket_i^B)$ for $j \in [2, 6, 10, 14]$, $\llbracket t[j] \rrbracket_i^B \leftarrow 3\text{-OR}(\llbracket d[j] \rrbracket_i^B, \llbracket d[j+1] \rrbracket_i^B, \llbracket d[j+2] \rrbracket_i^B)$ for $j \in [1, 5, 9, 13]$, and $\llbracket t[j] \rrbracket_i^B \leftarrow 4\text{-OR}(\llbracket d[j] \rrbracket_i^B, \llbracket d[j+1] \rrbracket_i^B, \llbracket d[j+2] \rrbracket_i^B, \llbracket d[j+3] \rrbracket_i^B)$ for $j \in [0, 4, 8, 12]$.
 - 3: P_i compute $\llbracket t'[j] \rrbracket_i^B = \llbracket t[j] \rrbracket_i^B$ for $j \in [3, 7, 11, 15]$ and compute $\llbracket t'[j] \rrbracket_i^B = \llbracket t[j] \rrbracket_i^B \oplus \llbracket t[j+1] \rrbracket_i^B$ for $j \in [0, 1, 2, 4, 5, 6, 8, 9, 10, 12, 13, 14]$.
 - 4: P_i locally compute $\llbracket w[j] \rrbracket_i^B = \bigoplus_{k=4j}^{4j+3} \llbracket t'[k] \rrbracket_i^B$ for $j \in [1, 2, 3]$.
 - 5: P_0 sets $\llbracket u[j] \rrbracket_0^B = 0$ for $j \in [0, \dots, 15]$ and P_1 sets $\llbracket u[j] \rrbracket_1^B = \llbracket d[j] \rrbracket_1^B$ for $j \in [0, \dots, 15]$.
 - 6: P_i parallelly compute $\llbracket v[j] \rrbracket_i^B \leftarrow 2\text{-AND}(\llbracket t'[j] \rrbracket_i^B, \llbracket u[j] \rrbracket_i^B)$ for $j \in [12, \dots, 15]$, $\llbracket v[j] \rrbracket_i^B \leftarrow 3\text{-AND}(\llbracket t'[j] \rrbracket_i^B, \llbracket u[j] \rrbracket_i^B, \neg \llbracket w[3] \rrbracket_i^B)$ for $j \in [8, \dots, 11]$, $\llbracket v[j] \rrbracket_i^B \leftarrow 4\text{-AND}(\llbracket t'[j] \rrbracket_i^B, \llbracket u[j] \rrbracket_i^B, \neg \llbracket w[2] \rrbracket_i^B, \neg \llbracket w[3] \rrbracket_i^B)$ for $j \in [4, \dots, 7]$, and $\llbracket v[j] \rrbracket_i^B \leftarrow 5\text{-AND}(\llbracket t'[j] \rrbracket_i^B, \llbracket u[j] \rrbracket_i^B, \neg \llbracket w[1] \rrbracket_i^B, \neg \llbracket w[2] \rrbracket_i^B, \neg \llbracket w[3] \rrbracket_i^B)$ for $j \in [0, \dots, 3]$.
 - 7: P_i locally compute $\llbracket z \rrbracket_i^B = \bigoplus_{\ell=0}^{15} \llbracket v[\ell] \rrbracket_i^B$.
 - 8: P_i compute $\llbracket z \rrbracket_i^B = \neg \llbracket z \rrbracket_i^B$.
 - 9: If $\llbracket x \rrbracket_1^A = 0$, then P_1 locally computes $\llbracket z \rrbracket_1^B = \llbracket z \rrbracket_1^B \oplus 1$.
 - 10: **return** $\llbracket z \rrbracket^B$.
-

Algorithm 4 Our Proposed B2A

Functionality: $\llbracket z \rrbracket^A \leftarrow \text{B2A}(\llbracket x \rrbracket^B)$
Ensure: $\llbracket z \rrbracket^A$, where $z = x$.

- 1: In pre-computation phase, the client randomly chooses $a, b \in \mathbb{Z}_{2^{16}}$, computes $c = ab$, chooses a randomness $r \in \mathbb{Z}_{2^{16}}$, and sets $(c_0, c_1) = (r, c - r)$. Then the client sends (a, c_0) and (b, c_1) to P_0 and P_1 , respectively.
 - 2: P_i ($i \in \{0, 1\}$) set $\llbracket x \rrbracket_i^A = \llbracket x \rrbracket_i^B$.
 - 3: P_0 computes $x' = \llbracket x \rrbracket_0^A - a$ and P_1 computes $x'' = \llbracket x \rrbracket_1^A - b$. Then they send them to each other.
 - 4: P_0 computes $\llbracket z \rrbracket_0^A = \llbracket x \rrbracket_0^A - 2(x'x'' + x'' \cdot a + c_0)$ and P_1 computes $\llbracket z \rrbracket_1^A = \llbracket x \rrbracket_1^A - 2(x' \cdot b + c_1)$.
 - 5: **return** $\llbracket z \rrbracket^A$.
-

posed the construction of B2A. In their protocol, we use a standard arithmetic multiplication protocol and need one communication round. In the setting of client-aided 2PC, however, B2A satisfies the condition that input party is equal to the computing party. Therefore, we can apply the techniques in Section 3.4 (arithmetic blinding) and construct more efficient B2A as in Algorithm 4: Although the number of communication rounds is the same as in [5], our protocol is more efficient. First, the data transfer in online phase is reduced from $2n$ -bits to n -bits. Moreover, the number of randomnesses we need in pre-computation is

reduced from five to three, and the data amount for sending from the client to P_0 and P_1 is reduced from $3n$ -bits to $2n$ -bits.

We can extend the above idea and obtain protocols like **BX2A**: $\llbracket b \rrbracket^B \times \llbracket x \rrbracket^A = \llbracket bx \rrbracket^A$, **BC2A**: $\llbracket b \rrbracket^B \times \llbracket c \rrbracket^B = \llbracket bc \rrbracket^A$, and **BCX2A**: $\llbracket b \rrbracket^B \times \llbracket c \rrbracket^B \times \llbracket x \rrbracket^A = \llbracket bcx \rrbracket^A$. These protocols are useful when we construct a round-efficient maximum value extraction protocol (and its variants) in Section 4.4.

BX2A: $\llbracket b \rrbracket^B \times \llbracket x \rrbracket^A = \llbracket bx \rrbracket^A$ We usually need to compute the multiplication of a boolean share $\llbracket b \rrbracket^B$ and an arithmetic one $\llbracket x \rrbracket^A$ (e.g., TLU in Section C.1, ReLU function in neural networks). We call this protocol **BX2A** in this paper. [24] proposed one-round **BX2A** under the $(2, 3)$ -replicated SS, such construction in 2PC has not been known. By almost the same idea as **B2A**, we can construct one-round **BX2A** in 2PC as follows:

1. P_i ($i \in \{0, 1\}$) set $\llbracket b \rrbracket_i^A = \llbracket b \rrbracket_i^B$.
2. P_0 sets $\llbracket b' \rrbracket_0^A = \llbracket b \rrbracket_0^B$ and $\llbracket b'' \rrbracket_0^A = 0$, and P_1 sets $\llbracket b' \rrbracket_1^A = 0$ and $\llbracket b'' \rrbracket_1^A = \llbracket b \rrbracket_1^B$.
3. P_i compute

$$\begin{aligned} \llbracket s \rrbracket_i^A &\leftarrow 2\text{-MULT}(\llbracket b \rrbracket_i^A, \llbracket x \rrbracket_i^A) \\ \llbracket t \rrbracket_i^A &\leftarrow 3\text{-MULT}(\llbracket b' \rrbracket_i^A, \llbracket b'' \rrbracket_i^A, \llbracket x \rrbracket_i^A). \end{aligned}$$

4. P_i computes $\llbracket z \rrbracket_i^A = \llbracket s \rrbracket_i^A - 2\llbracket t \rrbracket_i^A$.

Here, we denote this computation as $\llbracket bx - 2b_0b_1x \rrbracket^A$.

BC2A: $\llbracket b \rrbracket^B \times \llbracket c \rrbracket^B = \llbracket bc \rrbracket^A$ Almost the same idea as **BX2A**, we can compute $\llbracket b \rrbracket^B \times \llbracket c \rrbracket^B = \llbracket bc \rrbracket^A$ (**BC2A**) with one communication round. We use this protocol in **3-Argmax/3-Argmin** in Section 4.4. We can construct one-round **BC2A** by computing

$$\llbracket bc - 2b_0b_1 - 2c_0c_1 + 2b_0\bar{c}_0b_1\bar{c}_1 + 2\bar{b}_0\bar{c}_0\bar{b}_1\bar{c}_1 \rrbracket^A.$$

We need 2-MULT and 4-MULT for this protocol.

BCX2A: $\llbracket b \rrbracket^B \times \llbracket c \rrbracket^B \times \llbracket x \rrbracket^A = \llbracket bcx \rrbracket^A$ Almost the same idea as the above protocols, we can also compute $\llbracket b \rrbracket^B \times \llbracket c \rrbracket^B \times \llbracket x \rrbracket^A = \llbracket bcx \rrbracket^A$ (**BCX2A**) with one communication round. We use this protocol in **Max/Min** in Section 4.4. We can construct one-round **BC2A** by computing

$$\llbracket bcx - 2b_0b_1x - 2c_0c_1x + 2b_0\bar{c}_0b_1\bar{c}_1x + 2\bar{b}_0\bar{c}_0\bar{b}_1\bar{c}_1x \rrbracket^A.$$

We need 3-MULT and 5-MULT for this protocol.

Algorithm 5 Our Proposed 3-Max

Functionality: $\llbracket z \rrbracket^A \leftarrow \text{Max}(\llbracket \mathbf{x} \rrbracket^A)$
Ensure: $\llbracket z \rrbracket^A$, where z is the largest element in \mathbf{x} .

- 1: P_i ($i \in \{0, 1\}$) parallelly compute
 - $\llbracket c_{01} \rrbracket^B \leftarrow \text{Comparison}(\llbracket x[0] \rrbracket^A, \llbracket x[1] \rrbracket^A)$,
 - $\llbracket c_{02} \rrbracket^B \leftarrow \text{Comparison}(\llbracket x[0] \rrbracket^A, \llbracket x[2] \rrbracket^A)$, and
 - $\llbracket c_{12} \rrbracket^B \leftarrow \text{Comparison}(\llbracket x[1] \rrbracket^A, \llbracket x[2] \rrbracket^A)$.
 - 2: P_i compute $\llbracket c_{10} \rrbracket_i^B = \neg \llbracket c_{01} \rrbracket_i^B$, $\llbracket c_{20} \rrbracket_i^B = \neg \llbracket c_{02} \rrbracket_i^B$, and $\llbracket c_{21} \rrbracket_i^B = \neg \llbracket c_{12} \rrbracket_i^B$.
 - 3: P_i parallelly compute
 - $\llbracket t[0] \rrbracket_i^A \leftarrow \text{BCX2A}(\llbracket c_{10} \rrbracket_i^B, \llbracket c_{20} \rrbracket_i^B, \llbracket x[0] \rrbracket^A)$,
 - $\llbracket t[1] \rrbracket_i^A \leftarrow \text{BCX2A}(\llbracket c_{01} \rrbracket_i^B, \llbracket c_{21} \rrbracket_i^B, \llbracket x[1] \rrbracket^A)$, and
 - $\llbracket t[2] \rrbracket_i^A \leftarrow \text{BCX2A}(\llbracket c_{02} \rrbracket_i^B, \llbracket c_{12} \rrbracket_i^B, \llbracket x[2] \rrbracket^A)$.
 - 4: P_i compute $\llbracket z \rrbracket_i^A = \sum_{j=0}^2 \llbracket t[j] \rrbracket_i^A$.
 - 5: **return** $\llbracket z \rrbracket^A$.
-

4.4 The Maximum Value Extraction Protocol and Extensions

The maximum value extraction protocol $\text{Max}(\llbracket \mathbf{x} \rrbracket^A)$ outputs $\llbracket z \rrbracket^A$, where z is the largest value in \mathbf{x} . We first explain the case of Max for three elements (3-Max), which is used for computing edit distance, etc. We denote a j -th element of \mathbf{x} as $x[j]$; that is, $\mathbf{x} = [x[0], x[1], x[2]]$.

We start from a standard tournament-based construction. If the condition $x[0] < x[1]$ holds, $x' = x[1]$. Otherwise, $x' = x[0]$. By repeating the above procedure once more using $\llbracket x' \rrbracket^A$ and $\llbracket x[2] \rrbracket^A$, we can extract the maximum value among \mathbf{x} . In this strategy, we need 16 ($= (6 + 1 + 1) \times 2$) communication rounds, and 8 ($= (3 + 1) \times 2$) communication rounds even if we apply our three-round Comparison (in Section 4.2) and BX2A (in Section 4.3). This is mainly because we cannot parallelly execute Comparison . To solve this disadvantage, we first check the magnitude relationship for all elements using Comparison . Then we extract the maximum value. Based on these ideas, we show our 3-Max as in Algorithm 5: Although the computation costs obviously increased, this is four-round 3-Max by applying our Comparison and BCX2A .

Based on the above idea, we can also obtain the minimum value extraction protocol, argument of the maximum/minimum extraction protocols, and (argument of) the maximum/minimum value extraction protocols with $N(> 3)$ inputs. We show the construction of these protocols in Appendix C.3, Appendix C.4, and Appendix C.5, respectively.

5 Performance Evaluation

We demonstrate the practicality of our arithmetic/boolean gates and protocols. We implemented 2PC simulators and performed all benchmarks on a single laptop computer with Intel Core i7-6700K 4.00GHz and 64GB RAM. We implemented simulators using Python 3.7 with Numpy v1.16.2 and vectorized all gates/protocols. We assumed 10MB/s ($= 80000\text{bits/ms}$) bandwidth and 40ms

	pre-comp. time (ms)	online comp. time (ms)	# of comm. bits (bit)	data trans. time (ms)	# of comm. rounds	comm. latency (ms)	online total exec. time (ms)
2-AND	0.015	0.019	2	2.5×10^{-5}	1	40	40.0
	2.39	0.033	2×10^3	2.5×10^{-2}	1	40	40.1
	2439	19.4	2×10^6	25.0	1	40	84.4
3-AND	0.041	0.032	3	3.75×10^{-5}	1	40	40.0
	4.80	0.053	3×10^3	3.75×10^{-2}	1	40	40.1
	4899	33.1	3×10^6	37.5	1	40	110.6
4-AND	0.067	0.055	4	5.0×10^{-5}	1	40	40.1
	9.04	0.091	4×10^3	5.0×10^{-2}	1	40	40.1
	9383	62.8	4×10^6	50.0	1	40	152.8
5-AND	0.11	0.089	5	6.25×10^{-5}	1	40	40.1
	17.2	0.16	5×10^3	6.25×10^{-2}	1	40	40.2
	17700	111.7	5×10^6	62.5	1	40	214.2
6-AND	0.20	0.16	6	7.5×10^{-5}	1	40	40.2
	33.0	0.28	6×10^3	7.5×10^{-2}	1	40	40.4
	34059	203.0	6×10^6	75.0	1	40	318.0
7-AND	0.38	0.32	7	8.75×10^{-5}	1	40	40.3
	64.3	0.53	7×10^3	8.75×10^{-2}	1	40	40.6
	66123	370.8	7×10^6	87.5	1	40	498.3
8-AND	0.76	0.64	8	1.0×10^{-4}	1	40	40.6
	125.1	1.06	8×10^3	1.0×10^{-1}	1	40	41.2
	129553	700.7	8×10^6	100.0	1	40	840.7
9-AND	1.63	1.39	9	1.125×10^{-4}	1	40	41.4
	245.2	2.25	9×10^3	1.125×10^{-1}	1	40	42.4
	255847	1346	9×10^6	112.5	1	40	1498.5

Table 1. Evaluation on N -AND with 1(upper)/1000(middle)/1000000(lower) batch.

RTT latency as typical WAN settings, and calculate the data transfer time (DTT) and communication latency (CL) using these values. We adopted the client-aided model; that is, we assumed in our experiments that clients generate BTE in their local environment without using HE/OT.

5.1 Performance of Basic Gates

Here we show experimental results on N -AND. We set $N = [2, \dots, 9]$ and 1 to $10^6 (= 1000000)$ batch in our experiments. Here we show the experimental results on the cases of 1/1000/1000000 batch. The experimental results on other cases (10/100/10000/100000 batch) are in Appendix E. The results are as in Table 1 and Figure 1: We find (1) the pre-computation time, online computation time, and data transfer time are exponentially growing up with respect to N ; (2) the dominant part in online total execution time is WAN latency especially in the case of small batch. If we compute $N (> 2)$ -AND using multiple 2-AND gates, we need two or more communication rounds. Therefore, our scheme is especially suitable for the 2PC with relatively small batch (e.g., $\leq 10^5$) as it yields low WAN latency.

5.2 Performance of Our Protocols

Here we show experimental results on our proposed protocols (Equality, Comparison, and 3-Max). We implemented the baseline protocols [5] and our proposed ones in Section 4. Same as the evaluation of N -AND, we here show the results of our experiments over $\mathbb{Z}_{2^{32}}$ with 1/1000/1000000 batch in Table 2 and Figure 2

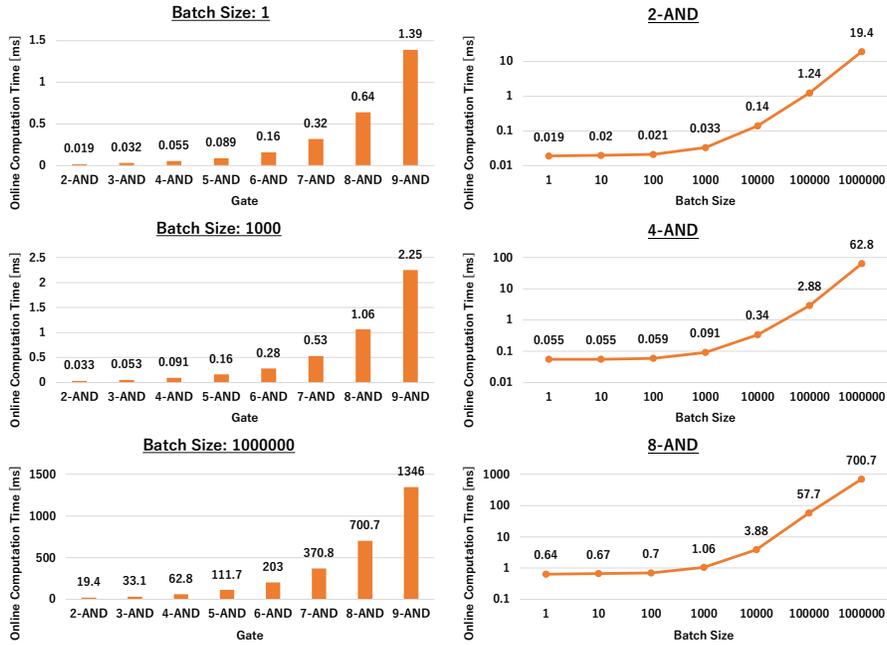


Fig. 1. Relations between N (fan-in number), batch size, and online computation time for N -AND: we show the relations between N and online computation time with 1/1000/1000000 batch (left), and show the relations between batch size and online computation time for 2/4/8-AND (right).

(relations between batch size and online execution time). Other results (protocols over $\mathbb{Z}_{2^{16}}$, $\mathbb{Z}_{2^{64}}$, and $\mathbb{Z}_{2^{32}}$ with other batch sizes) are in Appendix E. Same as the cases with N -AND, WAN latency is the dominant part of the online total execution time. In relatively small batch ($\leq 10^4$), all our protocols are faster than baseline ones in the online total execution time since ours require fewer communication rounds. For example in Comparison with 1 batch, we need more online computation time than the baseline one (0.54ms \rightarrow 2.1ms). However, communication costs of our Comparison are smaller than baseline one (the number of communication rounds: 7 \rightarrow 3, the number of communication bits: 970 \rightarrow 712). As a result, our Comparison is 56.1% faster than baseline one (280.6ms \rightarrow 122.1ms) in our WAN settings. As already mentioned, our protocols are not suitable for a (extremely) large batch since the computation cost is larger than baseline ones.

5.3 Application: Privacy-Preserving (Exact) Edit Distance

We implemented a privacy-preserving edit distance protocol using our protocols (Equality, B2A, and 3-Min). Unlike many previous works on approximate edit distance (e.g., [32]), here we consider the exact edit distance. We computed an

	pre-comp. time (ms)	online comp. time (ms)	# of comm. bits (bit)	data trans. time (ms)	# of comm. rounds	comm. latency (ms)	online total exec. time (ms)
Equality (1 batch)	0.15 0.76	0.18 0.52	62 38	7.75×10^{-4} 4.75×10^{-4}	5 2	200 80	200.2 80.5
Comparison (1 batch)	1.5 3.9	0.54 2.1	970 712	1.21×10^{-2} 8.9×10^{-3}	7 3	280 120	280.6 122.1
3-Max (1 batch)	3.1 9.7	1.2 2.3	2196 3960	2.75×10^{-2} 4.95×10^{-2}	18 4	720 160	721.2 162.3
Equality (10^3 batch)	74.7 500.5	0.61 1.1	62×10^3 38×10^3	0.78 0.48	5 2	200 80	201.4 80.9
Comparison (10^3 batch)	1398 2745	8.25 11.6	970×10^3 712×10^3	12.1 8.9	7 3	280 120	300.4 140.5
3-Max (10^3 batch)	2891 8635	17.5 36.3	2196×10^3 3960×10^3	27.5 49.5	18 4	720 160	765.0 245.8
Equality (10^6 batch)	77574 500617	761.4 1233	62×10^6 38×10^6	780 480	5 2	200 80	1741 1793
Comparison (10^6 batch)	1445847 2799437	13895 20748	970×10^6 712×10^6	12100 8900	7 3	280 120	26275 29768
3-Max (10^6 batch)	2956155 8571664	28252 69935	2196×10^6 3960×10^6	27500 49500	18 4	720 160	56472 119595

Table 2. Evaluation of our protocols over $\mathbb{Z}_{2^{32}}$ for $1/10^3/10^6$ batches. In each cell, we show our experimental results on the baseline (upper) and ours (lower).



Fig. 2. Relations between batch size and online computation/execution time of the protocols over $\mathbb{Z}_{2^{32}}$.

edit distance between two length- L genome strings (S_0 and S_1) via standard dynamic programming (DP). It appears four characters in the strings; that is, A, T, G, and C. In DP-matrix, we fill the cell $x[i][j]$ by the following rule:

$$x[i][j] = 3 - \text{Min}([x[i-1][j] + 1, x[i][j-1] + 1, x[i-1][j-1] + e])$$

Here, $e = 0$ if the condition $S_0[i] = S_1[j]$ holds, and otherwise $e = 1$. We can compute e using Equality (two rounds) and B2A (one round). To reduce the total online execution time, we calculate the edit distance as follows:

1. To reduce the total communication rounds, we parallelly compute e for all cells and store them in advance. Thanks to this procedure, we can avoid calculating e every time when we fill cells. We only need three communication rounds for this step.
2. Diagonal cells in DP-matrix are independent with each other. Therefore, we can parallelly compute these cells $x[d][0], x[d-1][1], \dots, x[0][d]$ (for each d) to reduce the communication rounds.

string length	pre-comp. time (s)	online comp. time (s)	data trans. time (s)	comm. latency (s)	online total exec. time (s)
4	0.04	0.01	4.0×10^{-4}	1.24	1.25
8	0.14	0.02	1.4×10^{-3}	2.52	2.54
16	0.57	0.04	5.7×10^{-3}	5.08	5.13
32	2.2	0.10	2.3×10^{-2}	10.2	10.3
64	8.1	0.22	9.2×10^{-2}	20.4	20.7
128	33.4	0.54	3.7×10^{-1}	40.9	41.8
256	135.7	1.5	1.5	84.9	84.9
512	534.1	4.8	5.9	163.8	174.5
1024	2262	16.0	23.4	327.6	367.0

Table 3. Experimental results of privacy-preserving exact edit distance with 2^ℓ -length two strings ($\ell = [2, \dots, 10]$).

By applying the above techniques, we can compute exact edit distance for two length- L strings with $3+4(2L-1) = (8L-1)$ communication rounds. We used the arithmetic shares and protocols over $\mathbb{Z}_{2^{16}}$ in our experiments. The experimental results are as in Table 3: As we can see from the experimental results, most of the online total execution time is occupied by the communication latency; that is, GC-based approaches may be much faster than SS-based one in WAN environments. However, if we would like to compute edit distances between many strings at the same time (e.g., the situation that the client has one string and the server has 1000 strings, and the client would like to compute edit distances between client’s string and all of server’s strings), SS-based approach will be much faster than GC-based one.

Acknowledgements. This work was partly supported by JST CREST JP-MJCR19F6 and the Ministry of Internal Affairs and Communications Grant Number 182103105.

References

1. Applebaum, B., Ishai, Y., Kushilevitz, E.: How to garble arithmetic circuits. *SIAM J. Comput.* **43**(2), 905–929 (2014). <https://doi.org/10.1137/120875193>
2. Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: High-throughput semi-honest secure three-party computation with an honest majority. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24–28, 2016. pp. 805–817 (2016). <https://doi.org/10.1145/2976749.2978331>
3. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Advances in Cryptology - CRYPTO ’91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11–15, 1991, Proceedings. pp. 420–432 (1991). https://doi.org/10.1007/3-540-46766-1_34
4. Ben-Efraim, A., Lindell, Y., Omri, E.: Optimizing semi-honest secure multiparty computation for the internet. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24–28, 2016. pp. 578–590 (2016). <https://doi.org/10.1145/2976749.2978347>
5. Bogdanov, D., Niitsoo, M., Toft, T., Willemsen, J.: High-performance secure multiparty computation for data mining applications. *Int. J. Inf. Sec.* **11**(6), 403–418 (2012). <https://doi.org/10.1007/s10207-012-0177-2>

6. Bost, R., Popa, R.A., Tu, S., Goldwasser, S.: Machine learning classification over encrypted data. In: 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015 (2015)
7. Bourse, F., Minelli, M., Minihold, M., Paillier, P.: Fast homomorphic evaluation of deep discretized neural networks. In: Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III. pp. 483–512 (2018). https://doi.org/10.1007/978-3-319-96878-0_17
8. Byali, M., Joseph, A., Patra, A., Ravi, D.: Fast secure computation for small population over the internet. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 677–694 (2018). <https://doi.org/10.1145/3243734.3243784>
9. Chida, K., Genkin, D., Hamada, K., Ikarashi, D., Kikuchi, R., Lindell, Y., Nof, A.: Fast large-scale honest-majority MPC for malicious adversaries. In: Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III. pp. 34–64 (2018). https://doi.org/10.1007/978-3-319-96878-0_2
10. Chida, K., Hamada, K., Ikarashi, D., Kikuchi, R., Kiribuchi, N., Pinkas, B.: An efficient secure three-party sorting protocol with an honest majority. Cryptology ePrint Archive, Report 2019/695 (2019)
11. Couteau, G., Peters, T., Pointcheval, D.: Encryption switching protocols. In: Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I. pp. 308–338 (2016). https://doi.org/10.1007/978-3-662-53018-4_12
12. Damgård, I., Fitzzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings. pp. 285–304 (2006). https://doi.org/10.1007/11681878_15
13. Demmler, D., Schneider, T., Zohner, M.: ABY - A framework for efficient mixed-protocol secure two-party computation. In: 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015 (2015)
14. Dessouky, G., Koushanfar, F., Sadeghi, A., Schneider, T., Zeitouni, S., Zohner, M.: Pushing the communication barrier in secure computation using lookup tables. In: 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017 (2017)
15. Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K.E., Naehrig, M., Wernsing, J.: Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In: Proceedings of the 33rd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016. pp. 201–210 (2016)
16. Goldreich, O.: The Foundations of Cryptography - Volume 2, Basic Applications. Cambridge University Press (2004)
17. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA. pp. 218–229 (1987). <https://doi.org/10.1145/28395.28420>
18. Hesamifard, E., Takabi, H., Ghasemi, M., Wright, R.N.: Privacy-preserving machine learning as a service. PoPETs **2018**(3), 123–142 (2018). <https://doi.org/10.1515/popets-2018-0024>

19. Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: GAZELLE: A low latency framework for secure neural network inference. In: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 1651–1669 (2018)
20. Kolesnikov, V., Sadeghi, A.R., Schneider, T.: How to combine homomorphic encryption and garbled circuits - improved circuits and computing the minimum distance efficiently. In: International Workshop on Signal Processing in the Encrypted Domain (SPEED'09) (2009)
21. Liu, J., Juuti, M., Lu, Y., Asokan, N.: Oblivious neural network predictions via minion transformations. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 619–631 (2017). <https://doi.org/10.1145/3133956.3134056>
22. Liu, X., Deng, R.H., Choo, K.R., Weng, J.: An efficient privacy-preserving outsourced calculation toolkit with multiple keys. *IEEE Trans. Information Forensics and Security* **11**(11), 2401–2414 (2016). <https://doi.org/10.1109/TIFS.2016.2573770>
23. Mohassel, P., Orobets, O., Riva, B.: Efficient server-aided 2pc for mobile phones. *PoPETs* **2016**(2), 82–99 (2016)
24. Mohassel, P., Rindal, P.: Aby^3 : A mixed protocol framework for machine learning. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 35–52 (2018). <https://doi.org/10.1145/3243734.3243760>
25. Mohassel, P., Rosulek, M., Zhang, Y.: Fast and secure three-party computation: The garbled circuit approach. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015. pp. 591–602 (2015). <https://doi.org/10.1145/2810103.2813705>
26. Mohassel, P., Zhang, Y.: Secureml: A system for scalable privacy-preserving machine learning. In: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017. pp. 19–38 (2017). <https://doi.org/10.1109/SP.2017.12>
27. Morita, H., Attrapadung, N., Teruya, T., Ohata, S., Nuida, K., Hanaoka, G.: Constant-round client-aided secure comparison protocol. In: Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II. pp. 395–415 (2018). https://doi.org/10.1007/978-3-319-98989-1_20
28. Nishide, T., Ohta, K.: Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In: Public Key Cryptography - PKC 2007, 10th International Conference on Practice and Theory in Public-Key Cryptography, Beijing, China, April 16-20, 2007, Proceedings. pp. 343–360 (2007). https://doi.org/10.1007/978-3-540-71677-8_23
29. Phong, L.T., Aono, Y., Hayashi, T., Wang, L., Moriai, S.: Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Trans. Information Forensics and Security* **13**(5), 1333–1345 (2018). <https://doi.org/10.1109/TIFS.2017.2787987>
30. Pinkas, B., Schneider, T., Tkachenko, O., Yanai, A.: Efficient circuit-based PSI with linear communication. In: Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III. pp. 122–153 (2019). https://doi.org/10.1007/978-3-030-17659-4_5

31. Riazi, M.S., Weinert, C., Tkachenko, O., Songhori, E.M., Schneider, T., Koushanfar, F.: Chameleon: A hybrid secure computation framework for machine learning applications. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018. pp. 707–721 (2018). <https://doi.org/10.1145/3196494.3196522>
32. Schneider, T., Tkachenko, O.: EPISODE: efficient privacy-preserving similar sequence queries on outsourced genomic databases. In: Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019. pp. 315–327 (2019). <https://doi.org/10.1145/3321705.3329800>
33. Yao, A.C.: How to generate and exchange secrets (extended abstract). In: 27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986. pp. 162–167 (1986). <https://doi.org/10.1109/SFCS.1986.25>
34. Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole - reducing data transfer in garbled circuits using half gates. In: Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II. pp. 220–250 (2015). https://doi.org/10.1007/978-3-662-46803-6_8
35. Zhu, R., Cassel, D., Sabry, A., Huang, Y.: NANOPI: extreme-scale actively-secure multi-party computation. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 862–879 (2018). <https://doi.org/10.1145/3243734.3243850>

A Semi-Honest Security

Here, we recall the simulation-based security notion in the presence of semi-honest adversaries (for 2PC) as in [16].

Definition 1. Let $f : (\{0, 1\}^*)^2 \rightarrow (\{0, 1\}^*)^2$ be a probabilistic 2-ary functionality and $f_i(\mathbf{x})$ denotes the i -th element of $f(\mathbf{x})$ for $\mathbf{x} = (x_0, x_1) \in (\{0, 1\}^*)^2$ and $i \in \{0, 1\}$; $f(\mathbf{x}) = (f_0(\mathbf{x}), f_1(\mathbf{x}))$. 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 $\mathbf{x} = (x_0, x_1) \in (\{0, 1\}^*)^2$ where $|x_0| = |x_1|$, denoted by $\text{VIEW}_i^\Pi(\mathbf{x})$, consists of $(x_i, r_i, m_{i,1}, \dots, m_{i,t})$, 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 an execution of Π on input \mathbf{x} is denoted as $\text{OUTPUT}^\Pi(\mathbf{x})$. 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

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

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

As described in [16], composition theorem for the semi-honest model holds; that is, any protocol is privately computed as long as its subroutines are privately computed.

B Correctness and Security of N -MULT/AND

B.1 Correctness of the Protocol

We have

$$\llbracket y \rrbracket_0 + \llbracket y \rrbracket_1 = \prod_{\ell=1}^N x'_\ell + \sum_{\emptyset \neq I \subseteq [1, N]} \left(\prod_{\ell \in [1, N] \setminus I} x'_\ell \right) a_I .$$

Since $a_I = \prod_{\ell \in I} a_{\{\ell\}}$, we have

$$\sum_{\emptyset \neq I \subseteq [1, N]} \left(\prod_{\ell \in [1, N] \setminus I} x'_\ell \right) a_I = (x'_1 + a_{\{1\}}) \cdots (x'_N + a_{\{N\}}) - x'_1 \cdots x'_N$$

therefore (by noting that $x'_\ell = x_\ell - a_{\{\ell\}}$)

$$\llbracket y \rrbracket_0 + \llbracket y \rrbracket_1 = \prod_{\ell=1}^N (x'_\ell + a_{\{\ell\}}) = \prod_{\ell=1}^N x_\ell .$$

Hence $\llbracket y \rrbracket_0$ and $\llbracket y \rrbracket_1$ form shares of $x_1 \cdots x_N$, as desired.

B.2 Security Proof of the Protocol

First we consider the security of the multiplication protocol against semi-honest P_0 (not colluding with Client). Let $(\llbracket x_\ell \rrbracket_0, \llbracket x_\ell \rrbracket_1)$ ($\ell = 1, \dots, N$) be fixed input shares, and let $\zeta \in \mathcal{M}$. We consider the conditional distribution of the view of P_0 for the case where the local output is $\llbracket y \rrbracket_0 = \zeta$.

The view of P_0 consists of $\llbracket a_I \rrbracket_0$ for $\emptyset \neq I \subseteq [1, N]$ and $\llbracket x'_\ell \rrbracket_1$ for $\ell = 1, \dots, N$ (note that the party uses no randomness in the protocol). Let α_I for $\emptyset \neq I \subseteq [1, N]$ and γ_ℓ for $\ell = 1, \dots, N$ be elements of \mathcal{M} . Let E denote the corresponding event that $\llbracket a_I \rrbracket_0 = \alpha_I$ holds for any $\emptyset \neq I \subseteq [1, N]$ and $\llbracket x'_\ell \rrbracket_1 = \gamma_\ell$ holds for any $\ell = 1, \dots, N$. By the construction of the protocol, if the event E occurs and moreover $\llbracket y \rrbracket_0 = \zeta$, then we have

$$\zeta = \alpha_{[1, N]} + \varphi_0((\alpha_I)_{I \neq [1, N]}, (\gamma_\ell)_\ell)$$

where

$$\varphi_0((\alpha_I)_{I \neq [1, N]}, (\gamma_\ell)_\ell) := \prod_{\ell=1}^N \gamma_\ell + \sum_{\substack{I \subseteq [1, N] \\ I \neq \emptyset, [1, N]}} \alpha_I \prod_{\ell \in [1, N] \setminus I} \gamma_\ell .$$

This implies that the conditional probability $\Pr[E \mid \llbracket y \rrbracket_0 = \zeta]$ is 0 if $\zeta \neq \alpha_{[1, N]} + \varphi_0((\alpha_I)_{I \neq [1, N]}, (\gamma_\ell)_\ell)$. We consider the other case where $\zeta = \alpha_{[1, N]} + \varphi_0((\alpha_I)_{I \neq [1, N]}, (\gamma_\ell)_\ell)$. Then the event E implies $\llbracket y \rrbracket_0 = \zeta$. Hence we have

$$\Pr[E \wedge \llbracket y \rrbracket_0 = \zeta] = \Pr[E] ,$$

therefore

$$\Pr[E \mid \llbracket y \rrbracket_0 = \zeta] = \Pr[E] / \Pr[\llbracket y \rrbracket_0 = \zeta] .$$

Now the event E occurs if and only if $\llbracket a_I \rrbracket_0 = \alpha_I$ for any $\emptyset \neq I \subseteq [1, N]$ and $\llbracket a_{\{\ell\}} \rrbracket_1 = \llbracket x_\ell \rrbracket_1 - \gamma_\ell$ for any $\ell = 1, \dots, N$. As the choices of $\llbracket a_I \rrbracket_0$'s and $\llbracket a_{\{\ell\}} \rrbracket_1$'s are uniformly random and independent, it follows that $\Pr[E]$ does not depend on α_I 's and γ_ℓ 's. On the other hand, we have $\Pr[\llbracket y \rrbracket_0 = \zeta] = 1/|\mathcal{M}|$ (independent of α_I 's and γ_ℓ 's), as for any choice of $\llbracket a_I \rrbracket_0$ for $I \neq \emptyset, [1, N]$ and of $\llbracket x'_\ell \rrbracket_1$ there is precisely one possibility of $\llbracket a_{[1, N]} \rrbracket_0$ that satisfies $\zeta = \llbracket a_{[1, N]} \rrbracket_0 + \varphi_0((\llbracket a_I \rrbracket_0)_{I \neq [1, N]}, (\llbracket x'_\ell \rrbracket_1)_\ell)$. Hence $\Pr[E \mid \llbracket y \rrbracket_0 = \zeta]$ is independent of α_I 's and γ_ℓ 's as well.

The argument above implies that, the distribution of the view of P_0 for fixed inputs and given local output $\llbracket y \rrbracket_0 = \zeta$ is the uniform distribution on the set of tuples $((\alpha_I)_I, (\gamma_\ell)_\ell)$ of elements of \mathcal{M} satisfying $\alpha_{[1, N]} + \varphi_0((\alpha_I)_{I \neq [1, N]}, (\gamma_\ell)_\ell) = \zeta$. The latter distribution can be sampled by freely choosing α_I for $I \neq [1, N]$ and γ_ℓ for $\ell = 1, \dots, N$ and then adjusting the value of $\alpha_{[1, N]}$. Hence, the view of P_0 is efficiently and perfectly simulatable, implying the security against semi-honest P_0 . The argument showing the security against semi-honest P_1 is similar (due to the aforementioned symmetry of the two parties in generating BTE), where we use the function φ_1 instead of φ_0 given by

$$\varphi_1((\alpha_I)_{I \neq [1, N]}, (\gamma_\ell)_\ell) := \sum_{\substack{I \subseteq [1, N] \\ I \neq \emptyset, [1, N]}} \alpha_I \prod_{\ell \in [1, N] \setminus I} \gamma_\ell .$$

This concludes the security proof of the protocol.

C Applications and Extensions of Our Protocols

C.1 Table Lookup

We can also obtain a round-efficient table lookup protocol TLU (or, 1-out-of- L oblivious transfer) using our **Equality**. As shown in previous results, TLU is useful function in secure computation (e.g., [14]). Here, we consider the table of arithmetic keys/values with size L (pairs of a j -th key K_j and a j -th value V_j for $j \in [0, \dots, L-1]$). We consider the situation that each computing party has shares of the table and a share of the index $\llbracket id \rrbracket_i^A$ and wants to obtain a share of the value V_j where $id = K_j$. To execute this protocol, we first check the equality of id and K_j for $j \in [0, \dots, L-1]$ via **Equality**. Then, we extract V_j using **BX2A** (in Section 4.3). We only need three communication rounds for this TLU.

C.2 Less-Than Comparison

To keep self-consistency of this paper, we explain how to construct a less-than comparison protocol **Comparison**($\llbracket x \rrbracket^A, \llbracket y \rrbracket^A$), which outputs $\llbracket z \rrbracket^B$, where $z = 1$ iff the condition $x < y$ holds. The high-level construction of this protocol is completely the same as in [5]; that is,

1. P_i ($i \in \{0, 1\}$) check whether the condition

$$\llbracket x \rrbracket_0^A \bmod 2^{n-1} + \llbracket x \rrbracket_1^A \bmod 2^{n-1} > 2^{n-1}$$

holds or not using **Overflow** and then compute $\llbracket x' \rrbracket^B = \llbracket \text{of}_x \rrbracket^B \oplus \llbracket \text{msb}_x \rrbracket^B$ (and the same for y and $d = x - y$, and obtain $\llbracket y' \rrbracket^B$ and $\llbracket d' \rrbracket^B$). Here, of_x denote the execution results of the above **Overflow** and msb_x denote the most significant bit of (binary expanded) x . We can extract the most significant bit of x , y , and d via the above operations.

2. P_i compute

$$\begin{aligned} \llbracket v \rrbracket^B &\leftarrow 2\text{-AND}(\llbracket x' \rrbracket^B \oplus \llbracket y' \rrbracket^B, \llbracket y' \rrbracket^B) \\ \llbracket w \rrbracket^B &\leftarrow 2\text{-AND}(\neg(\llbracket x' \rrbracket^B \oplus \llbracket y' \rrbracket^B), \llbracket d' \rrbracket^B). \end{aligned}$$

3. P_i compute $\llbracket z \rrbracket^B = \llbracket v \rrbracket^B \oplus \llbracket w \rrbracket^B$.

C.3 The Minimum Value Extraction Protocol

We can easily convert **Max** into **Min** by replacing the input order in step 1 in Algorithm 5 and obtain the minimum value extraction protocol for three elements (**3-Min**). We use this **3-Min** for executing privacy-preserving exact edit distance protocol in Section 5.3.

C.4 Argmax and Argmin

We can easily obtain Argmax/Argmin (by modifying Max/Min) as follows:

1. We replace $\llbracket t[j] \rrbracket^A \leftarrow \text{BCX2A}(\llbracket * \rrbracket^B, \llbracket ** \rrbracket^B, \llbracket x \rrbracket^A)$ in Algorithm 5 by $\llbracket t'[j] \rrbracket^A \leftarrow \text{BC2A}(\llbracket * \rrbracket^B, \llbracket ** \rrbracket^B)$.
2. P_i compute $\llbracket z \rrbracket_i^A = \sum_{j=0}^2 (j \cdot \llbracket t'[j] \rrbracket_i^A)$ in the step 4 in Algorithm 5, .

We can execute Argmax/Argmin with three communication rounds. We need fewer communication bits since we can avoid using BCX2A in these protocols. Note that in the above step 2, we need no interaction between computing parties since j is public.

C.5 N -Max/Min for $N > 3$

Even in the cases of Max/Min for four or more elements, we can construct round-efficient Max/Min with the same strategy as in Algorithm 5. However, there are two points of notice as follows:

1. In N -Max/Min, we need to (parallelly) execute Comparison $\frac{N(N-1)}{2}$ times. In the tournament-based strategy, we only need to execute Comparison for $\lceil \log N \rceil$ times; that is, in our protocols, computation costs and the amount of communication bits rapidly increase with respect to N .
2. For large N , we cannot directly use BCX2A (or BC2A). Although we can construct the protocol like $\llbracket bcdx \rrbracket^A = \llbracket b \rrbracket^B \times \llbracket c \rrbracket^B \times \llbracket d \rrbracket^B \times \llbracket x \rrbracket^A$, we can easily imagine that the computation costs we need for such a protocol increase drastically. To avoid such a disadvantage, we should split the step 3 in Algorithm 5 into some other protocols (e.g., $(N-1)$ -AND and BX2A). This means we need more communication rounds to execute N -Max/Min for large N .

D One-Round Overflow

In this section, we explain another construction of Overflow. Although we need more computation and data transfer than two-round Overflow in Section 4.2, we can compute the following Overflow with one communication round (for slightly small share spaces in practice).

Protocol Let $\chi[P]$ denote a bit that is 1 if the condition P holds and 0 otherwise. Let $\text{Overflow}(a, b; c) = \chi[a + b \geq c]$. Here, n_1 and n_2 are parameters with $n = n_1 + n_2$:

1. P_i ($i \in \{0, 1\}$) parses $\llbracket x \rrbracket_i^A = y_i \parallel z_i$ where y_i is the n_1 most significant bits of x_i and z_i is the n_2 least significant bits of x_i .
2. For each $a_1 = 1, \dots, 2^{n_1} - 1$,
 - (a) P_0 sets $\alpha_0^{(a_1;1)} \leftarrow \chi[y_0 = a_1]$ and $\alpha_0^{(a_1;2)} \leftarrow 0$.

- (b) P_1 sets $\alpha_1^{(a_1;1)} \leftarrow 0$ and $\alpha_1^{(a_1;2)} \leftarrow \chi[y_1 \geq 2^{n_1} - a_1]$.
 Let $\llbracket \alpha^{(a_1;j)} \rrbracket^{\mathbf{B}} = (\alpha_0^{(a_1;j)}, \alpha_1^{(a_1;j)})$ for $j = 1, 2$.
3. For each $a_2 = 1, \dots, 2^{n_2} - 1$ and $j = 0, \dots, n_1 - 1$, P_0 sets

$$\beta_0^{(a_2;j)} \leftarrow \begin{cases} y_0[j] & \text{if } y_0 \neq 0 \text{ and } z_0 = a_2, \\ 1 & \text{otherwise,} \end{cases}$$

where $y_0[j]$ denotes the j -th bit of y_0 . P_1 sets

$$\beta_1^{(a_2;j)} \leftarrow \begin{cases} y_1[j] & \text{if } y_1 \neq 0 \text{ and } z_1 \geq 2^{n_2} - a_2, \\ 1 & \text{otherwise.} \end{cases}$$

Let $\llbracket \beta^{(a_2;j)} \rrbracket^{\mathbf{B}} = (\beta_0^{(a_2;j)}, \beta_1^{(a_2;j)})$.

4. For each $a_3 = 1, \dots, 2^{n_2} - 1$,
- (a) P_0 sets $\gamma_0^{(a_3;1)} \leftarrow \chi[y_0 = 0]$, $\gamma_0^{(a_3;2)} \leftarrow \chi[y_0 = 2^{n_1} - 1]$, $\gamma_0^{(a_3;3)} \leftarrow \chi[z_0 = a_3]$, and $\gamma_0^{(a_3;4)} \leftarrow 0$.
- (b) P_1 sets $\gamma_1^{(a_3;1)} \leftarrow \chi[y_1 = 0]$, $\gamma_1^{(a_3;2)} \leftarrow \chi[y_1 = 2^{n_1} - 1]$, $\gamma_1^{(a_3;3)} \leftarrow 0$, and $\gamma_1^{(a_3;4)} \leftarrow \chi[z_1 \geq 2^{n_2} - a_3]$.
- Let $\llbracket \gamma^{(a_3;j)} \rrbracket^{\mathbf{B}} = (\gamma_0^{(a_3;j)}, \gamma_1^{(a_3;j)})$ for $j = 1, 2, 3, 4$.
5. Two parties execute the followings in parallel: For each $a_1 = 1, \dots, 2^{n_1} - 1$, compute

$$\llbracket b_1^{(a_1)} \rrbracket \leftarrow 2\text{-AND}(\llbracket \alpha^{(a_1;1)} \rrbracket^{\mathbf{B}}, \llbracket \alpha^{(a_1;2)} \rrbracket^{\mathbf{B}})$$

by using 2-AND. For each $a_2 = 1, \dots, 2^{n_2} - 1$, compute

$$\llbracket b_2^{(a_2)} \rrbracket^{\mathbf{B}} \leftarrow n_1\text{-AND}(\llbracket \beta^{(a_2;0)} \rrbracket^{\mathbf{B}}, \llbracket \beta^{(a_2;1)} \rrbracket^{\mathbf{B}}, \dots, \llbracket \beta^{(a_2;n_1-1)} \rrbracket^{\mathbf{B}})$$

by using n_1 -AND. For each $a_3 = 1, \dots, 2^{n_2} - 1$, compute

$$\llbracket b_3^{(a_3)} \rrbracket^{\mathbf{B}} \leftarrow 4\text{-AND}(\llbracket \gamma^{(a_3;1)} \rrbracket^{\mathbf{B}}, \llbracket \gamma^{(a_3;2)} \rrbracket^{\mathbf{B}}, \llbracket \gamma^{(a_3;3)} \rrbracket^{\mathbf{B}}, \llbracket \gamma^{(a_3;4)} \rrbracket^{\mathbf{B}})$$

by using 4-AND.

6. P_i locally compute

$$\llbracket d \rrbracket_i^{\mathbf{B}} \leftarrow \bigoplus_{a_1=1}^{2^{n_1}-1} \llbracket b_1^{(a_1)} \rrbracket_i^{\mathbf{B}} \oplus \bigoplus_{a_2=1}^{2^{n_2}-1} \llbracket b_2^{(a_2)} \rrbracket_i^{\mathbf{B}} \oplus \bigoplus_{a_3=1}^{2^{n_2}-1} \llbracket b_3^{(a_3)} \rrbracket_i^{\mathbf{B}}.$$

Then P_i output the share $\llbracket d \rrbracket_i^{\mathbf{B}}$.

All the steps except Step 5 can be locally executed by each party. Hence, in total, only 1 round of communication is required which is spent during Step 5, where $(2^{n_1} - 1)$ 2-ANDs, $(2^{n_2} - 1)$ n_1 -ANDs, and $(2^{n_2} - 1)$ 4-ANDs are performed in parallel. For example, when $n = 15$ and $(n_1, n_2) = (8, 7)$, these are 255 2-ANDs, 127 8-ANDs, and 127 4-ANDs.

Correctness First, we note that an overflow occurs modulo 2^n for (x_0, x_1) if and only if, either an overflow occurs modulo 2^{n_1} for (y_0, y_1) , or $y_0 + y_1 = 2^{n_1} - 1$ and an overflow occurs modulo 2^{n_2} for (z_0, z_1) . As the two events are disjoint, it follows that

$$\begin{aligned} & \text{Overflow}(x_0, x_1; 2^n) \\ &= \text{Overflow}(y_0, y_1; 2^{n_1}) \oplus (\chi[y_0 + y_1 = 2^{n_1} - 1] \wedge \text{Overflow}(z_0, z_1; 2^{n_2})) . \end{aligned}$$

Moreover, we have

$$\chi[y_0 + y_1 = 2^{n_1} - 1] = \bigwedge_{j=0}^{n_1-1} (y_0[j] \oplus y_1[j]) ,$$

therefore $\text{Overflow}(x_0, x_1; 2^n)$ is the XOR of $\text{Overflow}(y_0, y_1; 2^{n_1})$ and

$$\left(\bigwedge_{j=0}^{n_1-1} (y_0[j] \oplus y_1[j]) \right) \wedge \text{Overflow}(z_0, z_1; 2^{n_2}) . \quad (1)$$

For the term $\text{Overflow}(y_0, y_1; 2^{n_1})$, we note that the overflow occurs if and only if there is a (in fact, unique) $a_1 = 1, \dots, 2^{n_1} - 1$ satisfying that $y_0 = a_1$ and $a_1 + y_1 \geq 2^{n_1}$ (i.e., $y_1 \geq 2^{n_1} - a_1$). These $2^{n_1} - 1$ events are all disjoint. In the protocol, the bit $\alpha^{\langle a_1; 1 \rangle}$ is 1 if and only if $y_0 = a_1$, and the bit $\alpha^{\langle a_1; 2 \rangle}$ is 1 if and only if $y_1 \geq 2^{n_1} - a_1$. Therefore, we have

$$\text{Overflow}(y_0, y_1; 2^{n_1}) = \bigoplus_{a_1=1}^{2^{n_1}-1} \alpha^{\langle a_1; 1 \rangle} \wedge \alpha^{\langle a_1; 2 \rangle} = \bigoplus_{a_1=1}^{2^{n_1}-1} b_1^{\langle a_1 \rangle} .$$

The same argument implies that, the bit in Eq.(1) is equal to

$$\begin{aligned} & \left(\bigwedge_{j=0}^{n_1-1} (y_0[j] \oplus y_1[j]) \right) \wedge \bigoplus_{a_2=1}^{2^{n_2}-1} \chi[z_0 = a_2] \wedge \chi[z_1 \geq 2^{n_2} - a_2] \\ &= \bigoplus_{a_2=1}^{2^{n_2}-1} \left(\left(\bigwedge_{j=0}^{n_1-1} (y_0[j] \oplus y_1[j]) \right) \wedge \chi[z_0 = a_2] \wedge \chi[z_1 \geq 2^{n_2} - a_2] \right) . \end{aligned} \quad (2)$$

To decrease the depth of the circuit in Eq.(2), we consider to let P_0 modify the bits $y_0[j]$ in a way that the AND-term becomes 0 if z_0 (known to P_0) is not equal to a_2 . Now observe that, unless $y_1 = 0$, at least one of the bits $y_1[j]$ is 1, therefore the AND-term would become 0 if all bits $y_0[j]$ were 1. Accordingly, instead of $y_0[j]$, we use a bit $y'_0[j]_{a_2}$ that is $y_0[j]$ if $z_0 = a_2$ and is 1 if $z_0 \neq a_2$. Then we have

$$\begin{aligned} & \left(\bigwedge_{j=0}^{n_1-1} (y_0[j] \oplus y_1[j]) \right) \wedge \chi[z_0 = a_2] \wedge \chi[z_1 \geq 2^{n_2} - a_2] \\ &= \left(\bigwedge_{j=0}^{n_1-1} (y'_0[j]_{a_2} \oplus y_1[j]) \right) \wedge \chi[z_1 \geq 2^{n_2} - a_2] \end{aligned}$$

unless $y_1 = 0$. Similarly, we let P_1 modify the bits $y_1[j]$ in a way that the AND-term becomes 0 if z_1 (known to P_1) is smaller than $2^{\ell_2} - a_2$. Namely, instead of $y_1[j]$, we use a bit $y'_1[j]_{a_2}$ that is $y_1[j]$ if $z_1 \geq 2^{n_2} - a_2$ and is 1 otherwise. Then the same argument implies that

$$\left(\bigwedge_{j=0}^{n_1-1} (y'_0[j]_{a_2} \oplus y_1[j]) \right) \wedge \chi[z_1 \geq 2^{n_2} - a_2] = \bigwedge_{j=0}^{n_1-1} (y'_0[j]_{a_2} \oplus y'_1[j]_{a_2})$$

unless $y_0 = 0$. Summarizing, the bit in Eq.(1) is equal to

$$\bigoplus_{a_2=1}^{2^{n_2}-1} \left(\bigwedge_{j=0}^{n_1-1} (y'_0[j]_{a_2} \oplus y'_1[j]_{a_2}) \right)$$

unless $y_0 = 0$ or $y_1 = 0$.

Now we want to adjust the computation result in the case where $y_0 = 0$ or $y_1 = 0$. Before doing that, we modify the computation further in order to simplify the situation: for $i = 0, 1$, we change the bits $y'_i[j]_{a_2}$ in a way that it always becomes 1 when $y_i = 0$. The resulting bit is equal to $\beta_i^{\langle a_2; j \rangle}$ in the protocol, and the corresponding computation result

$$\bigoplus_{a_2=1}^{2^{n_2}-1} \left(\bigwedge_{j=0}^{n_1-1} (\beta_0^{\langle a_2; j \rangle} \oplus \beta_1^{\langle a_2; j \rangle}) \right) = \bigoplus_{a_2=1}^{2^{n_2}-1} \left(\bigwedge_{j=0}^{n_1-1} \beta^{\langle a_2; j \rangle} \right) = \bigoplus_{a_2=1}^{2^{n_2}-1} b_2^{\langle a_2 \rangle} \quad (3)$$

is still equal to the bit in Eq.(1) unless $y_0 = 0$ or $y_1 = 0$. On the other hand, when $y_0 = 0$ or $y_1 = 0$, the bit in Eq.(3) is equal to 0, as now one of the two vectors $(\beta_i^{\langle a_2; 0 \rangle}, \dots, \beta_i^{\langle a_2; n_1-1 \rangle})$ ($i = 0, 1$) is $(1, 1, \dots, 1)$ while the other has at least one component being 1. When $y_0 = y_1 = 0$, the bit in Eq.(1) is also equal to 0 and hence is equal to the bit in Eq.(3) as desired. From now, we consider the other case where precisely one of y_0 and y_1 is equal to 0; in the protocol, this is equivalent to $\gamma_0^{\langle a_3; 1 \rangle} \oplus \gamma_1^{\langle a_3; 1 \rangle} = 1$, i.e., $\gamma^{\langle a_3; 1 \rangle} = 1$. Under the condition, the bit in Eq.(1) becomes 1 if and only if the other y_i which is not equal to 0 is equal to $(11 \dots 1)_2 = 2^{n_1} - 1$ (i.e., $\gamma^{\langle a_3; 2 \rangle} = \gamma_0^{\langle a_3; 2 \rangle} \oplus \gamma_1^{\langle a_3; 2 \rangle} = 1$ in the protocol) and $\text{Overflow}(z_0, z_1; 2^{n_2}) = 1$. By expanding the bit $\text{Overflow}(z_0, z_1; 2^{n_2})$ in the same way as the aforementioned case of $\text{Overflow}(y_0, y_1; 2^{n_1})$, it follows that the bit in Eq.(1) is equal to

$$\bigoplus_{a_3=1}^{2^{n_2}-1} \gamma^{\langle a_3; 1 \rangle} \wedge \gamma^{\langle a_3; 2 \rangle} \wedge \gamma^{\langle a_3; 3 \rangle} \wedge \gamma^{\langle a_3; 4 \rangle} = \bigoplus_{a_3=1}^{2^{n_2}-1} b_3^{\langle a_3 \rangle}$$

under the current condition. Note that the bit above is 0 when the current condition (i.e., precisely one of y_0 and y_1 is 0) is not satisfied.

Summarizing the arguments, the bit in Eq.(1) is equal to

$$\bigoplus_{a_2=1}^{2^{n_2}-1} b_2^{\langle a_2 \rangle} \oplus \bigoplus_{a_3=1}^{2^{n_2}-1} b_3^{\langle a_3 \rangle}$$

	pre-comp. time (ms)	online comp. time (ms)	# of comm. bits (bit)	data trans. time (ms)	# of comm. rounds	comm. latency (ms)	online total exec. time (ms)
2-AND	0.037	0.020	2×10^1	2.5×10^{-4}	1	40	40.0
	0.24	0.021	2×10^2	2.5×10^{-3}	1	40	40.0
	23.2	0.14	2×10^4	2.5×10^{-1}	1	40	40.4
	243.1	1.2	2×10^5	2.5	1	40	43.7
3-AND	0.085	0.033	3×10^1	3.75×10^{-4}	1	40	40.0
	0.50	0.035	3×10^2	3.75×10^{-3}	1	40	40.0
	46.3	0.21	3×10^4	3.75×10^{-1}	1	40	40.6
	489.8	1.9	3×10^5	3.75	1	40	45.7
4-AND	0.15	0.055	4×10^1	5.0×10^{-4}	1	40	40.1
	0.94	0.059	4×10^2	5.0×10^{-3}	1	40	40.1
	89.3	0.34	4×10^4	5.0×10^{-1}	1	40	40.8
	929.0	2.9	4×10^5	5.0	1	40	47.9
5-AND	0.26	0.096	5×10^1	6.25×10^{-4}	1	40	40.1
	1.8	0.098	5×10^2	6.25×10^{-3}	1	40	40.1
	168.6	0.58	5×10^4	6.25×10^{-1}	1	40	41.2
	1763	5.0	5×10^5	6.25	1	40	51.3
6-AND	0.49	0.17	6×10^1	7.5×10^{-4}	1	40	40.2
	3.4	0.18	6×10^2	7.5×10^{-3}	1	40	40.2
	327.8	1.0	6×10^4	7.5×10^{-1}	1	40	41.8
	3379	13.1	6×10^5	7.50	1	40	60.6
7-AND	0.96	0.32	7×10^1	8.75×10^{-4}	1	40	40.3
	6.5	0.34	7×10^2	8.75×10^{-3}	1	40	40.3
	644.6	2.0	7×10^4	8.75×10^{-1}	1	40	42.9
	6564	27.4	7×10^5	8.75	1	40	76.2
8-AND	1.9	0.67	8×10^1	1.0×10^{-3}	1	40	40.7
	12.8	0.70	8×10^2	1.0×10^{-2}	1	40	40.7
	1274	3.9	8×10^4	1.0	1	40	44.9
	12868	57.7	8×10^5	10.0	1	40	107.7
9-AND	3.9	1.4	9×10^1	1.125×10^{-3}	1	40	41.4
	25.3	1.5	9×10^2	1.125×10^{-2}	1	40	41.5
	2538	9.9	9×10^4	1.125	1	40	51.0
	25515	121.7	9×10^5	11.25	1	40	173.0

Table 4. Evaluation of N -AND with 10(top)/100(second from the top)/10000(third from the top)/100000(bottom) batch.

in any case, therefore we have

$$\begin{aligned}
\text{Overflow}(x_0, x_1; 2^n) &= \bigoplus_{a_1=1}^{2^{n_1}-1} b_1^{(a_1)} \oplus \bigoplus_{a_2=1}^{2^{n_2}-1} b_2^{(a_2)} \oplus \bigoplus_{a_3=1}^{2^{n_3}-1} b_3^{(a_3)} \\
&= d
\end{aligned}$$

as desired. This completes the proof of correctness for the protocol.

E Other Experimental Results

Here we show the experimental results of our gates and protocols we omit in Section 5.

	pre-comp. time (ms)	online comp. time (ms)	# of comm. bits (bit)	data trans. time (ms)	# of comm. rounds	comm. latency (ms)	online total exec. time (ms)
Equality	5.3	0.52	20×10^1	4.75×10^{-3}	2	80	80.5
	50.3	0.59	20×10^2	4.75×10^{-2}	2	80	80.6
	5003	5.6	20×10^4	4.75	2	80	90.4
	50051	101.2	20×10^5	4.75×10^1	2	80	228.7
Comparison	28.1	2.2	712×10^1	8.9×10^{-2}	3	120	122.3
	266.9	3.2	712×10^2	8.9×10^{-1}	3	120	124.1
	27810	138.2	712×10^4	8.9×10^1	3	120	347.2
	282130	2171	712×10^5	8.9×10^2	3	120	3181
3-Max	83.5	2.7	3960×10^1	2.75×10^{-1}	4	160	163.0
	841.3	5.6	3960×10^2	2.75	4	160	168.4
	86345	631.5	3960×10^4	2.75×10^2	4	160	1067
	863023	7121	3960×10^5	2.75×10^3	4	160	10031

Table 5. Evaluation of our protocols over $\mathbb{Z}_{2^{32}}$ with 10(top)/100(second from the top)/10000(third from the top)/100000(bottom) batch.

	pre-comp. time (ms)	online comp. time (ms)	# of comm. bits (bit)	data trans. time (ms)	# of comm. rounds	comm. latency (ms)	online total exec. time (ms)
Equality	0.17	0.17	20	2.5×10^{-4}	2	80	80.2
	0.59	0.17	20×10^1	2.5×10^{-3}	2	80	80.2
	4.6	0.19	20×10^2	2.5×10^{-2}	2	80	80.2
	46.0	0.36	20×10^3	2.5×10^{-1}	2	80	80.6
	447.3	1.8	20×10^4	2.5	2	80	84.3
	4591	30.5	20×10^5	2.5×10^1	2	80	135.5
45982	376.9	20×10^6	2.5×10^2	2	80	706.9	
Comparison	0.95	0.98	280	3.5×10^{-3}	3	120	121.0
	6.4	0.98	280×10^1	3.5×10^{-2}	3	120	121.0
	58.3	1.3	280×10^2	3.5×10^{-1}	3	120	121.6
	598.9	3.7	280×10^3	3.5	3	120	127.2
	5995	30.5	280×10^4	3.5×10^1	3	120	185.5
	60666	555.7	280×10^5	3.5×10^2	3	120	1026
607179	5349	280×10^6	3.5×10^3	3	120	8969	
3-Max	2.5	1.2	1752	2.19×10^{-2}	4	160	161.2
	19.8	1.3	1752×10^1	2.19×10^{-1}	4	160	161.5
	189.7	2.4	1752×10^2	2.19	4	160	164.6
	1947	12.0	1752×10^3	2.19×10^1	4	160	193.9
	20121	216.0	1752×10^4	2.19×10^2	4	160	595.0
	199728	2415	1752×10^5	2.19×10^3	4	160	4765
1976891	22868	1752×10^6	2.19×10^4	4	160	44928	

Table 6. Evaluation of our protocols over $\mathbb{Z}_{2^{16}}$ with 1 to 10^6 batch (from the top to the bottom).

	pre-comp. time (ms)	online comp. time (ms)	# of comm. bits (bit)	data trans. time (ms)	# of comm. rounds	comm. latency (ms)	online total exec. time (ms)
Equality	2.5	1.4	72	9.0×10^{-4}	2	80	81.4
	12.7	1.5	72×10^1	9.0×10^{-3}	2	80	81.5
	112.9	1.8	72×10^2	9.0×10^{-2}	2	80	81.9
	1152	4.7	72×10^3	9.0×10^{-1}	2	80	85.6
	11404	53.4	72×10^4	9.0	2	80	142.4
	114999	658.8	72×10^5	9.0×10^1	2	80	828.8
	1156086	7862	72×10^6	9.0×10^2	2	80	8842
Comparison	22.0	5.8	1900	2.38×10^{-2}	3	120	125.8
	183.3	6.3	1900×10^1	2.38×10^{-1}	3	120	126.5
	1819	11.6	1900×10^2	2.38	3	120	134.0
	18894	74.4	1900×10^3	2.38×10^1	3	120	218.2
	186987	975.0	1900×10^4	2.38×10^2	3	120	1333
	1861936	13418	1900×10^5	2.38×10^3	3	120	15918
	19098870	245178	1900×10^6	2.38×10^4	3	120	269098
	58.5	6.3	9348	1.17×10^{-1}	4	160	166.4
3-Max	543.7	8.3	9348×10^1	1.17	4	160	169.5
	5487	23.7	9348×10^2	1.17×10^1	4	160	195.4
	56608	270.0	9348×10^3	1.17×10^2	4	160	547.0
	564780	3440	9348×10^4	1.17×10^3	4	160	4770
	5721530	42420	9348×10^5	1.17×10^4	4	160	54280
	—	—	—	—	—	—	—
	—	—	—	—	—	—	—

Table 7. Evaluation of our protocols over $\mathbb{Z}_{2^{64}}$ with 1 to 10^6 batch (from the top to the bottom). We could not execute 3-Max with 1000000 batch in our experiments because of the memory shortage.