Constant-Round Client-Aided Secure Comparison Protocol

Hiraku Morita[1(✉)], Nuttapong Attrapadung[1], Tadanori Teruya[1], Satsuya Ohata[1], Koji Nuida[2], and Goichiro Hanaoka[1]

[1] AIST, Tokyo, Japan
{hiraku.morita,n.attrapadung,tadanori.teruya,satsuya.ohata,
hanaoka-goichiro}@aist.go.jp
[2] The University of Tokyo, Tokyo, Japan
nuida@mist.i.u-tokyo.ac.jp

Abstract. We present an improved constant-round secure two-party protocol for integer comparison functionality, which is one of the most fundamental building blocks in secure computation.

Our protocol is in the so-called *client-server model*, which is utilized in real-world MPC products such as Sharemind, where any number of clients can create shares of their input and distribute to the servers who then jointly compute over the shares and return the shares of result to the client. In the *client-aided* client-server model, as mentioned briefly by Mohassel and Zhang (S&P'17), a client further generates and distributes some necessary correlated randomness to servers. Such correlated randomness admits efficient protocols since otherwise servers have to jointly generate randomness by themselves, which can be inefficient.

In this paper, we improve the state-of-the-art constant-round comparison protocols by Damgård et al. (TCC'06) and Nishide and Ohta (PKC'07) in the client-aided model. Our techniques include identifying correlated randomness in these comparison protocols. Along the way, we also use tree-based techniques for a building block, which deviate from the above two works. Our proposed protocol requires only 5 communication rounds, regardless of the bit length of inputs. This is at least 5 times fewer rounds than existing protocols. We implement our secure comparison protocol in C++. Our experimental results show that this low-round complexity benefits in low-latency networks such as WAN.

Keywords: Multi-party computation · Client-server model
Client-aided method · Less-than comparison · Constant rounds
GMW secret sharing

1 Introduction

Multi-party computation (MPC) is a powerful cryptographic tool often used to achieve privacy-preserving applications such as secure data mining. In general, MPC enables a set of N parties to jointly compute a function, say f, of

J. Lopez et al. (Eds.): ESORICS 2018, LNCS 11099, pp. 395–415, 2018.
https://doi.org/10.1007/978-3-319-98989-1_20

their private inputs. More precisely, the N parties, each holding private input x_i (for $i = 1, \ldots, N$), are able to compute the output $F = f(x_1, \ldots, x_N)$ without having to reveal their private inputs x_i. The security of MPC guarantees that a party i will learn nothing about the others' inputs, namely, x_j for all j not equal to i, except the information that can be derived from the output F and his/her own input x_i.

In this paper, we focus on secret-sharing based MPC [13], as opposed to other approaches such as garbled-circuit [24] or (fully) homomorphic encryption. Comparing among these, secret-sharing based MPC normally admits low computational cost and low bandwidth, while it generally requires more round communications. Constructing *round-efficient* protocols is thus one of the main goals for secret-sharing based MPC.

Secure Comparison Protocols. In this paper, we study integer comparison functionality, which has been considered one of the most fundamental building blocks for MPC since a seminal paper by Yao [24] introduced the Millionaires' problem, which itself is the starting point for researches on MPC. It has many applications that include auctions, machine learning, data clustering, statistical analysis, applications involving sorting, finding minimum/maximum, to name just a few. Secure comparison protocols have many variants (*cf.* [2]); in order to be able to flexibly use them as building blocks in larger applications, it is imperative to consider the variant with *shared inputs and shared output*. More precisely, the inputs to the protocol are shares of x and shares of y, while the output comprises shares of bit b which indicates the result of comparing $x < y$ (note that it is sufficient w.l.o.g. to consider less-than functionality). Throughout the paper, we consider this variant unless stated otherwise.

Despite being such a central functionality, inefficiency of comparison protocols is often a bottleneck for the applications listed above. Such inefficiency inherently stems from the fact that on one hand, applications are *arithmetic* computations; while, on the other hand, computing integer comparison is a *bit string* operation by nature, and protocols that compute such bit decompositions often require $\log n$ rounds, where n is the bit length of inputs.

A breakthrough result was proposed by Damgård et al. [7], who came up with the first secret-sharing based comparison protocol that admits *constant rounds*. Their protocol can be based on any linear secret-sharing based MPC that has multiplication protocol, and require 44 rounds of multiplication protocols (as counted in [15]). When including overall communication such as sharing or revealing phases (see more discussion on this in Sect. 4), in this paper, it can be counted to 79 overall rounds. Nishide and Ohta [15] proposed an improved protocol that has 15 rounds of multiplication protocols, and as counted in this paper, has 28 overall rounds. In this paper, we will improve these constants albeit working in the *client-aided client-server* model.

MPC in the Client-Server Model. In pushing MPC towards real-world usages, the setting of so-called *client-server model* for MPC has recently been largely motivated not only by recent researches including Araki *et al.* [1] but also by commercial-grade MPC products such as Sharemind system by Cybernetica.

Table 1. Comparison of LessThan protocols in the number of rounds, total communication, and estimated total online execution time

Secure LessThan	Round	Total communication* (number of field elements)	Total online time† (ms)
Damgård et al. [7]	79	$176n \log n + 80n \log \log n + 70n$	5688
Nishide-Ohta [15]	28	$96n + 120 \log n + 4$	2016
Damgård et al. + Client-aided	70	$144n \log n + 64n \log \log n + 52n$	5040
Nishide-Ohta + Client-aided	14	$36n + 48 \log n + 7$	1008
Ours	6	$12n^2 + 25$	432
Ours (round reduced)	5	$12n^2 + 301$	360

*For total communication of Damgård et al. [7] and its client-aided version, only dominant terms are shown here (for simplicity). More details can be found in Appendix B.
†Total time is estimated in a WAN setting where the network delay is 72 ms.

In such a model, there are N servers and an unbounded number of clients, say t. Each client provides his input x_i by secret-sharing it to the N servers, who will then jointly compute in secure manner over these input shares and return the output $f(x_1, \ldots, x_t)$ to clients.[1] This setting is suitable in real-world business innovation as the MPC engine run by servers can be thought of "as a service". In particular, each client only participates at the input phase and simply waits for the output. A program for client can thus contain only a simple and lightweight computation, namely, the secret sharing procedure, and hence makes it possible to be easily employed (*e.g.,* as a tiny script program in web browsers).

Client-Aided Client-Server Model. In the *client-aided* client-server model, as mentioned briefly by Mohassel and Zhang [14], a client, who distributes shares of its input to servers, further generates and distributes some necessary *correlated randomness* to servers. Such correlated randomness will be used by the N servers for running a protocol among them. This is for the purpose of better efficiency, since otherwise servers would have to jointly generate randomness by themselves, which can be inefficient. The only downside for this model is the restriction that any server is assumed to not collude with the client who generates such correlated randomness; doing so would break security. But this restriction seems reasonable already in the client-server business model, as a server would normally have no incentive to collude to a client.[2]

[1] As a side note, this setting can be considered as $(N + t)$-party MPC, where the N parties have no input.

[2] This, however, depends on applications. Nevertheless, in most cases, a company (a server) might have to be worried more about losing its credit if the fact that it colludes with a client to obtain other client's secret is somehow exposed.

1.1 Our Contribution

Our main contribution is an efficient secure comparison (LessThan) protocol in the client-aided client-server model with two servers (and with an unbounded number of clients). It improves upon the state-of-the-art secure comparison protocols that achieve a constant round complexity. We show a comparison for the number of communication rounds in Table 1, which also shows the total communication and an estimated total time for executing a protocol in a WAN setting (see below). The number of overall rounds for our protocol is 5, which is considerably much lower than the previous schemes (at least 5 times fewer rounds than Nishide-Ohta [15], which requires 28 overall rounds). We also implement our secure comparison protocol in C++. Our experimental results show that this low-round complexity benefits in low-latency networks such as WAN (also see below).

Our Techniques. Our protocol is based loosely on the previous protocols of [15], which is, in turn, based on [7]. Our techniques for reducing rounds consist of the following strategies (described only in a high-level overview):

- We first note that [15] uses the LSB (least significant bit) protocol as a building block. While [7,15] can use any linear secret sharing with multiplication protocol, we use a specific secret-sharing scheme, namely, the 2-out-of-2 sharing scheme. Note that such a secret scheme is the base for the original MPC by Goldreich *et al.* [13], which we denote the GMW scheme. This enables us to construct the LSB protocol based on a comparison protocol with *plain inputs* and shared output, called PlainLessThan protocol.
- We then construct a protocol for PlainLessThan by using a tree-based structure called *dyadic range*, similarly to [2]. This has two advantages. First, such a structure admits parallel computations (hence, is suitable for constant-round protocols). Second, each computation is multi-fan-in AND, which we can construct a constant-round protocol.
- We finally construct a constant-round multi-fan-in AND protocol using the protocol proposed also in [7]. This is the point where we utilize the client-aided setting so that the correlated randomness generation phase is entirely computed by a client. We identify the necessary correlated randomness by removing redundancy in [7]. Our client-aided protocol is more sophisticated than the one in [14], which considers correlated randomness for only a simple multiplication protocol (such randomness is called Beaver multiplication triple [3] in the literature); ours is for the whole multi-fan-in AND protocol.

More details for intuition on our building blocks can be found in Sect. 3.

Better Total-Time Efficiency in WAN. While achieving less rounds, our protocol requires larger asymptotic complexity in total communication: ours is $O(n^2)$, versus $O(n \log n)$ and $O(n)$ in [7,15], resp., as shown in Table 1. However, when considering concrete real-world parameters and large-delay networks like WAN (Wide Area Network), this does not matter since the total time for transmitting data of any amount up to its capacity will be roughly the same. More

precisely, in WAN, we can set the transmission bandwidth to 9 MB/s and the network delay to 72 ms, as done in [14]. Hence, in one round, we can transmit any amount of data up to 9 MB/s × 72 ms = 648 KB, in roughly the same amount of time (72 ms). For our protocol, when considering $n = 32$ bits, the total transmitted data has about $12n^2 + 301 = 12589$ field elements; each element has 32 bits, hence the total data has only 402848 bits (50 KB), which is already less than the capacity of 648 KB. Moreover, the local computation time would contribute only negligible time compared to the network delay (see Sect. 5). Therefore, the total (online) time to run the protocol is indeed simply about the one-round time (72 ms) times the number of rounds, as shown in Table 1. We note also that, thanks to the client-aided method, the offline time is kept small compared to the online time (see Table 2). More details can also be found in Appendix B.

1.2 Related Work

Secure comparison protocols have been widely studied since Yao [24] introduced the Millionaire's protocol. Research on secure comparison protocols have a vast literature, *e.g.,* [4–9,11,15,21,22,24], and we would like to point the reader to an excellent survey published relatively recently in 2015 by Veugen *et al.* [23] for a detailed overview, while we briefly mention some more related ones here. As Veugen *et al.* [23] pointed out, secret sharing based secure comparisons [6,11,15] have an advantage in online phase in comparison with garbled circuit based protocols and homomorphic encryption based protocols. Attrapadung *et al.* [2] categorized various secure comparison protocols regarding their input/output forms. Damgård *et al.* [7] proposed a constant-round secure comparison scheme and Nishide-Ohta [15] developed the idea to construct fewer rounds secure comparison protocol, on which our protocol is based. We note that, in this paper, we count the round complexity in a strict sense: the communication rounds of revealing or sharing are also included (while, in most of previous papers, only those of multiplication protocols are counted). See more in Sect. 4. This somewhat leads to more round complexities than those in original papers. In subsequent works to [15], some other optimizations have been introduced based on the assumption that the compared values are restricted to be less than $\frac{p-1}{2}$ [16–18]. (To free up this restriction, the number of rounds would increase.) Reistad [16] claims that the online round complexity is 2; however, the actual round complexity (in our strict counting) seems to be much greater since similar sub-protocols to those in [15] are used. Their main advantage is, nevertheless, the total linear communication. While our focus is on reducing communication rounds to sufficiently small constant, there exist also logarithmic-round secure comparison protocols in literature (*e.g.,* [10,20]); our sub-protocols in Sect. 3 might be applicable to reduce the communication rounds in these cases too.

2 Preliminaries and Settings

In this section, we introduce notation and terminology. The general notion for a multi-party protocol to *compute* a function f and to *privately compute* a function f in the semi-honest model follows from the standard definition (*e.g.,* [12]).

As a basic terminology throughout the paper, we let p be an odd prime and n be the bit length of p. We represent elements in the prime field $\mathbb{F}_p$ as $\{0, \ldots, p-1\}$.

Syntax for Secret Sharing. An N-out-of-N secret sharing scheme over $\mathbb{F}_p$ consists of two algorithms: Share and Reveal. Share takes as input $x \in \mathbb{F}_p$, and outputs $(\llbracket x \rrbracket_1, \ldots, \llbracket x \rrbracket_N) \in \mathbb{F}_p^N$, where the bracket notation $\llbracket x \rrbracket_i$ denotes the share of the i-th party. We denote $\llbracket x \rrbracket = (\llbracket x \rrbracket_1, \ldots, \llbracket x \rrbracket_N)$ as their shorthand. Reveal takes as input $\llbracket x \rrbracket$, and outputs x.

Client-Server Model. We describe the setting for secret-sharing based MPC in the client-server model (similarly to *e.g.,* [1]) as follows. We assume that there exists N servers and an unbounded number of clients, say t. We assume that there exists a secure channel between any client and any server, and among any two servers (Note that a secure channel among clients are not needed).

Let S be an N-out-of-N secret sharing scheme over $\mathbb{F}_p$. We say that a protocol Π computes a function $f : \mathbb{A}^t \to \mathbb{B}$ in the client-server model with a secret sharing scheme S if Π proceeds as follows.

1. In the first pass, each client j (for $j \in [1, t]$) creates shares of its input $a_j \in \mathbb{A}$ as $\llbracket a_j \rrbracket = (\llbracket a_j \rrbracket_1, \ldots, \llbracket a_j \rrbracket_N) \leftarrow \mathsf{Share}(a_j)$. It then distributes $\llbracket a_j \rrbracket_i$ to the server i (for $i \in [1, N]$).
2. All the N servers jointly compute f over their shares. More precisely, in this joint protocol, the input from the server i is $(\llbracket a_1 \rrbracket_i, \ldots, \llbracket a_t \rrbracket_i)$. Let $b = f(a_1, \ldots, a_t)$. The output for the server i in this joint protocol is the share $\llbracket b \rrbracket_i$. We abuse the notation of f and write this protocol as
$$\llbracket b \rrbracket \leftarrow f(\llbracket a_1 \rrbracket, \ldots, \llbracket a_t \rrbracket).$$
3. In the final pass, each server i (for $i \in [1, N]$) returns $\llbracket b \rrbracket_i$ to all the clients. Each client can recover b by $\mathsf{Reveal}(\llbracket b \rrbracket)$.

Note that such a protocol setting is a specific case of $(N + t)$-party protocols, where N parties among these do not have input, and t parties among these participate only the first pass (for sending) and the final pass (for receiving). Therefore, the security notion in the semi-honest model follows from the standard notion of private computation (*e.g.,* [12]).

Client-Aided Client-Server Model. The client-aided setting (similarly to [14]) further specializes the above setting by allowing the following:

- A fixed client, w.l.o.g. say client number 1 (but could be any), will additionally send an auxiliary input aux_i to the server i (for all $i \in [1, N]$). The distribution of auxiliary inputs can be done offline or at the same time as the first pass in the client-server model described above. We denote $\mathsf{aux} = (\mathsf{aux}_1, \ldots, \mathsf{aux}_N)$.
- In the joint computation for f, each server i can input its auxiliary input aux_i. We write this protocol as $\llbracket b \rrbracket \leftarrow f(\llbracket a_1 \rrbracket, \ldots, \llbracket a_t \rrbracket; \mathsf{aux})$, where we also often omit explicitly writing aux when the context is clear.

We assume that the client that generates auxiliary inputs is honest and does not collude with any servers. As a remark, this setting can be considered as the trusted initializer model [19] where the "trusted initializer" in our case is one of the t clients.

Our Setting: Two-Server GMW Scheme. In this paper, we consider two servers, that is, $N = 2$. Hence, in particular, we only allow the adversary to corrupt only one server. We use the standard 2-out-of-2 secret sharing scheme defined by

- $\mathsf{Share}(x)$: randomly choose $r \in \mathbb{F}_p$ and let $[\![x]\!]_1 = r$ and $[\![x]\!]_2 = x - r$.
- $\mathsf{Reveal}([\![x]\!]_1, [\![x]\!]_2)$: output $[\![x]\!]_1 + [\![x]\!]_2$.

We note that this is the secret sharing scheme used in the original MPC by Goldreich, Micali, and Widgerson [13], hence we often call it the GMW-style two-party secret sharing scheme. In this scheme, we have protocols for fundamental operations: $\mathsf{ADD}(x, y) := x + y$ and $\mathsf{MULT}(x, y) := xy$ as follows:

- $[\![z]\!] \leftarrow \mathsf{ADD}([\![x]\!], [\![y]\!])$ can be done locally by simply adding its own share on x and on y.
- $[\![w]\!] \leftarrow \mathsf{MULT}([\![x]\!], [\![y]\!])$ can be done in various ways. We will use the standard method based on Beaver multiplication triples [3]. Such a triple consists of $\mathsf{aux}_1 = (a_1, b_1, c_1)$ and $\mathsf{aux}_2 = (a_2, b_2, c_2)$ such that $(a_1+a_2)(b_1+b_2) = c_1+c_2$. In particular, we can use the client-aided method to let a client generate and distribute aux_1 and aux_2 to the two servers, respectively.

We abuse notations and write the ADD protocol simply as $[\![z]\!] \leftarrow [\![x]\!] + [\![y]\!]$, and MULT protocols simply as $[\![w]\!] \leftarrow [\![x]\!] \cdot [\![y]\!]$. Note that multiplication with constant c can use the ADD protocol, and we write $c[\![x]\!]$.

3 Our Secure Comparison Protocol

In this section, we present our protocol for computing the less-than comparison functionality, $\mathsf{LessThan}$. It consists of various sub-protocols, which might be of independent interest in their own right, that we also present in this section. These consist of the following.

- MULT^*: multi-input multiplication functionality.
- AND^*: multi-fan-in AND functionality.
- $\mathsf{PlainEqual}$: equality test functionality with *plain* inputs.
- $\mathsf{PlainLessThan}$: less-than comparison functionality with *plain* inputs.
- $\mathsf{WrapAround}$: a functionality for testing if the addition of the two shares (in the integers, without modulo p) is more than p or not (wrapping around p or not).
- LSB: least-significant-bit functionality.
- $\mathsf{HalfTest}$: a functionality for testing if a (shared) input is less than $p/2$ or not.

We remark that all of these functionality except two have *shared inputs* and *shared output*. (Definitions for each functionality will be provided below.) The only two exceptions are PlainEqual and PlainLessThan, where inputs consist of plain values that are private to each party.

Outline of Our Protocol for LessThan. We will use the functionality for HalfTest to construct LessThan, and LSB to construct HalfTest in exactly the same manner as in [15]. To construct a protocol for LSB, we will use WrapAround, which is then constructed based on PlainLessThan. This is different to the construction of LSB in [15]. Our protocol for PlainLessThan is based on binary tree structure, which admits parallel computation (and hence use small constant rounds of communication). This is somewhat related to the protocol in [2], with the difference that here our protocol uses secret sharing, while [2] uses homomorphic encryption. PlainLessThan uses PlainEqual as a subroutine. PlainEqual then uses the multi-fan-in AND, namely, AND^*, as a subroutine. AND^* is then based on the multi-input multiplication, namely, MULT^*, in a similar manner to [7]. Finally, MULT^* is constructed based on using correlated randomness produced by the aiding party.

3.1 Multi-input Multiplication Protocol (MULT^*)

We first describe the smallest building block (besides ADD, MULT), namely, the multi-input multiplication functionality, MULT^*. Its definition is the computation protocol as follows.

$$([\![c_1]\!], [\![c_2]\!], \ldots, [\![c_\ell]\!]) \leftarrow \mathsf{MULT}^*([\![x_1]\!], \ldots, [\![x_\ell]\!]),$$

where we define $c_i := x_1 x_2 \cdots x_i$ for all $i \in [1, \ell]$.

Intuition/Approach. Our protocol follows the basic mechanism of the protocol for MULT^* in [7]. The protocol of [7] lets parties collaborate and produce shares of random elements, say t_j, and its *inverse*, namely, t_j^{-1}. Such procedures are somewhat costly. Our idea is to gather and optimize all these correlated randomness elements in one place and let it be generated by an aiding party. For example, the "chaining" like $q_j := t_{j-1} t_j^{-1}$ will be pre-computed. Moreover, the protocol of [7] can be used for any linear secret sharing scheme with MULT protocol. When using a MULT protocol that uses multiplication triples, the correlated randomness for MULT will become redundant with those t_j, t_j^{-1}. We eliminate these redundancy by generating multiplication triples directly over q_j defined as above, and not an independent randomness.

Correlated Randomness for MULT^*. The aiding party locally pre-computes the following:

1. For all $j \in [0, \ell]$, pick $t_j \xleftarrow{\$} \mathbb{F}_p^\times$, and also compute its inverse, t_j^{-1}.
2. For all $j \in [1, \ell]$, define $q_j := t_{j-1} t_j^{-1}$, $z_j := t_0^{-1} t_j$, and also pick $a_j \in \mathbb{F}_p^\times$.
3. Set the correlated randomness for P_1 and P_2 to be the following random shares:

$$\mathsf{CR}_\ell := ([\![a_j]\!], [\![q_j]\!], [\![a_j q_j]\!], [\![z_j]\!])_{i \in [1,\ell]}\,. \tag{1}$$

Our Protocol for MULT^*. We show our protocol for MULT^* in Algorithm 1.

Algorithm 1. MULT^* Protocol

Functionality: $([\![c_1]\!], [\![c_2]\!], \ldots, [\![c_\ell]\!]) \leftarrow \mathsf{MULT}^*([\![x_1]\!], \ldots, [\![x_\ell]\!])$.
Input: Arithmetic shared values $[\![x_1]\!], \ldots, [\![x_\ell]\!]$ over $\mathbb{F}_p$ and an integer ℓ.
Auxiliary Input: CR_ℓ in Eq. (1).
Output: $([\![c_1]\!], [\![c_2]\!], \ldots, [\![c_\ell]\!])$ over $\mathbb{F}_p$.

1: For all $j \in [1, \ell]$, in parallel, compute and reveal

$$[\![x_j]\!] - [\![a_j]\!].$$

Hence, each party learns $x_j - a_j$.

2: For all $j \in [1, \ell]$, in parallel, compute and reveal

$$[\![d_j]\!] \leftarrow (x_j - a_j) \cdot [\![q_j]\!] + [\![a_j q_j]\!].$$

Hence, each party learns $d_j := (x_j - a_j)q_j + a_j q_j = x_j q_j$.

3: For all $j \in [2, \ell]$, (locally) compute

$$[\![c_j]\!] \leftarrow d_1 \cdots d_j \cdot [\![z_j]\!]. \tag{2}$$

Correctness/Security. The protocol correctness can be shown by verifying Eq. (2), which is indeed correct since $d_1 \cdots d_j \cdot z_j = (x_1 q_1) \cdots (x_j q_j) \cdot t_0^{-1} t_j = x_1 \cdots x_j \cdot (t_0 t_1^{-1}) \cdot (t_1 t_2^{-1}) \cdots (t_{j-1} t_j^{-1}) \cdot t_0^{-1} t_j x_1 \cdots x_j = c_j$. We sketch an argument for security as follows. We observe that the only points where potential information leak may occur are the revealing of $x_j - a_j$ and of $x_j q_j$. However, a_j and q_j are used only once, and hence they act as one-time pad to x_j (additively and multiplicatively, respectively). More precisely, in $x_j q_j$, the value x_j is multiplicatively blinded by t_j^{-1}. Note also that a_j and q_j are available to the parties as shares, hence no information on a_j and q_j leaks either. To prove more formally from this argument in the standard simulator-based notion (*e.g.,* [12]), we just define a simulator that simply simulates the view in the step 1 and 2 of the protocol by random elements. This simulated view is indistinguishable from the real protocol view exactly by the one-time use of a_j and q_j.

A Special Case: Power. For further use, we also define a special case of MULT^* where all the inputs x_j are the same, say x. It thus computes powers of an element x. For formality, we write $([\![x]\!], [\![x^2]\!], \ldots, [\![x^\ell]\!]) \leftarrow \mathsf{Power}([\![x]\!], \ell)$.

3.2 Multi-fan-in AND (AND^*)

We next describe the multi-fan-in AND functionality and a protocol for it below. This is defined by $[\![y_1 \wedge \cdots \wedge y_m]\!] \leftarrow \mathsf{AND}^*([\![y_1]\!], \ldots, [\![y_m]\!])$.

Intuition/Approach. We construct this protocol based on MULT^* (or more precisely, Power) using exactly the approach for *symmetric function* evaluation in [7]. In such a function, the output depends only on the number of 1's in its input. Hence, it can be interpreted as a function with input $\sum_{j=1}^{m} y_j$. (To exclude it being 0 which is problematic, we will add 1 to it, similarly to [7,15].)

This function can be constructed via Lagrange interpolation. In the case of the multi-fan-in AND functionality, its corresponding interpolated function (with coefficient $c_k \in \mathbb{F}_p$) is defined by

$$p_m(x) = \frac{1}{m!} \prod_{j=1}^{m} (x - j) \bmod p =: \sum_{k=0}^{m} c_k x^k. \tag{3}$$

Our Protocol for AND^*. We show our protocol for AND^* in Algorithm 2.

Algorithm 2. AND^* Protocol

Functionality: $[\![y_1 \wedge \cdots \wedge y_m]\!] \leftarrow \mathsf{AND}^*([\![y_1]\!], \ldots, [\![y_m]\!])$.
Input: Arithmetic shared values $[\![y_1]\!], \ldots, [\![y_m]\!]$ over $\mathbb{F}_p$ where $y_j \in \{0, 1\}$ for all $j \in [1, m]$.
Auxiliary Input: CR_m.
Output: $[\![y_1 \wedge \cdots \wedge y_m]\!]$ over $\mathbb{F}_p$.
1: Compute (locally) for $[\![x]\!] \leftarrow 1 + \sum_{j=1}^{m} [\![y_j]\!]$.
2: Compute $([\![x]\!], [\![x^2]\!], \ldots, [\![x^m]\!]) \leftarrow \mathbf{Power}([\![x]\!], m)$.
3: Compute (locally) and output $[\![v]\!] \leftarrow c_0 + \sum_{k=1}^{m} c_k [\![x^k]\!]$, where the coefficients c_k's are as in Eq.(3).

Correctness/Security. We can verify that $v = y_1 \wedge \cdots \wedge y_m$ as follows: First, the AND function is a symmetric function and thus the output depends only on the value $x := 1 + \sum_{j=1}^{m} y_j$. In particular, we have that

$$y_1 \wedge \cdots \wedge y_m = \begin{cases} 0 & \text{if } x \in [1, m] \\ 1 & \text{if } x = m + 1 \end{cases}.$$

The Lagrange interpolation of the polynomial defined on these $m + 1$ points are indeed the polynomial in Eq. (3). That is, $y_1 \wedge \cdots \wedge y_m = \sum_{k=0}^{m} c_k x^k$, and hence $v = y_1 \wedge \cdots \wedge y_m$, as required. As for security, it holds straightforwardly since we only call Power as a subroutine.

3.3 Equality Test with Plain Inputs ($\mathsf{PlainEqual}$)

We next describe the equality test functionality with *plain* inputs (and shared output). This is defined by $[\![\delta]\!] \leftarrow \mathsf{PlainEqual}(x, y)$ where $\delta = 1$ if $x = y$, and $\delta = 0$ otherwise.

Intuition/Approach. We construct this protocol in straightforward way. To confirm equality of x and y, we check if the i-th bit of x, namely x_i, equals the i-th bit of y, namely y_i. Instead of sharing each bit, we let the share of the other party be 0 so as to save communication.[3]

[3] This does not leak any private information as long as addition (or subtraction) of shared values is executed soon afterward as in Step 3 of Algorithm 3, where subtraction $1 - [\![x_i]\!] - [\![y_i]\!]$ is computed first and then its squared value is computed.

Our Protocol for PlainEqual**.** We show our protocol for PlainEqual in Algorithm 3.

Algorithm 3. PlainEqual Protocol

Functionality: $[\![\delta]\!] \leftarrow \mathsf{PlainEqual}(x, y)$, where $\delta = 1$ if $x = y$; $\delta = 0$, otherwise.
Input: Cleartexts $x, y \in \mathbb{F}_p$
Output: Arithmetic shared value $[\![\delta]\!]$
1: Parse $x = x_{n-1} \,\|\, x_{n-2} \,\|\, \ldots \,\|\, x_0$.
2: Parse $y = y_{n-1} \,\|\, y_{n-2} \,\|\, \ldots \,\|\, y_0$.
3: P_1 sets $[\![x_i]\!]_1 \leftarrow x_i$ and $[\![y_i]\!]_1 \leftarrow 0$ for $i \in [0, n-1]$.
4: P_2 sets $[\![x_i]\!]_2 \leftarrow 0$ and $[\![y_i]\!]_2 \leftarrow y_i$ for $i \in [0, n-1]$.
5: Compute $[\![v_i]\!] \leftarrow (1 - [\![x_i]\!] - [\![y_i]\!])^2$ for $i \in [0, n-1]$.
6: Compute $[\![\delta]\!] \leftarrow \mathsf{AND}^*([\![v_0]\!], [\![v_1]\!], \ldots [\![v_{n-1}]\!])$.
7: **return** $[\![\delta]\!]$.

Correctness/Security. If $x = y$, the i-th bit of x matches the i-th bit of y, that is, $x_i = y_i$ for $i \in [0, n-1]$, where $x = x_{n-1} \,\|\, x_{n-2} \,\|\, \ldots \,\|\, x_0$ and $y = y_{n-1} \,\|\, y_{n-2} \,\|\, \ldots \,\|\, y_0$ for $x_i, y_i \in \mathbb{Z}_2$. We set $[\![v_i]\!] \leftarrow (1 - [\![x_i]\!] - [\![y_i]\!])^2$ for $i \in [0, n-1]$. This value is 1 if $x_i = y_1$, and 0 otherwise. Now, we obtain that $x = y$ if and only if $v_i = 1$ for all $i \in [0, n-1]$.

$$\mathsf{AND}^*([\![v_0]\!], [\![v_1]\!], \ldots, [\![v_{n-1}]\!]) = \begin{cases} 0 & \text{if } \sum_{i=0}^{n-1} v_i \in [0, n-1] \\ 1 & \text{if } \sum_{i=0}^{n-1} v_i = n. \end{cases}$$

As for security, it holds straightforwardly since we only call AND^* as a subroutine.

3.4 Less-Than Comparison with Plain Inputs (PlainLessThan)

We describe the less-than comparison with plain inputs and a protocol for it below. This is defined by $[\![\delta]\!] \leftarrow \mathsf{PlainLessThan}(x, y)$ where the inputs are $x, y \in [0, p-1]$, and the output bit is $\delta = (x < y)$.[4]

Algorithm 4. PlainLessThan Protocol

Functionality: $[\![\delta]\!] \leftarrow \mathsf{PlainLessThan}(x, y)$, such that $\delta = (x < y)$
Input: Cleartext $x \in [0, p-1]$ from P_1, $y \in [0, p-1]$ from P_2
Output: Arithmetic shared value $[\![\delta]\!]$ over $\mathbb{F}_p$
1: P_1 sets $R = [x + 1, 2^n - 1]$.
2: P_1 computes $\{(i, a_i)\} \leftarrow \mathsf{rangeEnc}(R)$; $W_R \leftarrow \{i \mid \exists a \text{ s.t. } (i, a) \in \mathsf{rangeEnc}(R)\}$.
3: P_1 sets $a_j = 2^n - 1$ for all $j \in [0, n]$ s.t. $j \notin W_R$.
4: P_2 computes $\{(i, b_i)\} \leftarrow \mathsf{pointEnc}(y)$.
5: Compute $[\![d_i]\!] = \mathsf{PlainEqual}(a_i, b_i)$ for all $i \in [0, n]$.
6: $[\![\delta]\!] \leftarrow \sum_{i=0}^{n} [\![d_i]\!]$.
7: **return** $[\![\delta]\!]$.

Intuition/Approach. We construct a protocol for PlainLessThan based on binary-tree-based approach called *dyadic range* in a similar manner to [2]. The

[4] For a statement C, we denote $(C) = 1$ if C is true, and 0 otherwise;.

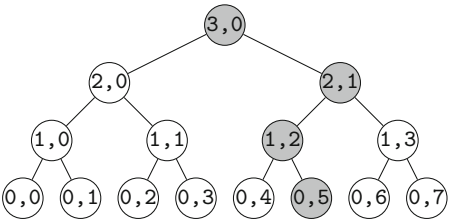

Fig. 1. Example of point encoding for $x = 5$. Here, $\mathsf{pointEnc}(5) = \{(0,5),(1,2),(2,1),(3,0)\}$.

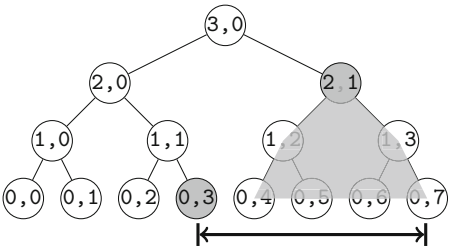

Fig. 2. Example of range encoding for $x = 2$, which defines the range $R = [3,7]$. Here, $\mathsf{rangeEnc}([3,7]) = \{(0,3),(2,1)\}$.

main idea of this tree-based approach is that when the inputs are plain, we can *directly* "encode" them to a data structure that is suitable for comparison in parallel. This encoding method is called *range and point encoding* in [2]. At the core of this approach is the equality test functionality over plain inputs. For this equality test, [2] uses additive homomorphic encryption. On the other hand, we construct this functionality by secret sharing, which is more computationally efficient; this is the main difference to [2]. As described above, our equality test essentially uses multi-fan-in AND as a building block.

Range/Point Encoding. We use a similar terminology from [2], which we recall here. Recall first that n is the bit length of p, that is, $n = \lceil \log_2 p \rceil$. Hence, in particular, $x, y \leq p-1 < 2^n - 1$. Let $\mathbb{T}_{2^n}$ be a complete binary tree whose leaves correspond to integers from 0 to $2^n - 1$. Let $\mathbb{S}_{2^n}$ be the set of all nodes in the tree $\mathbb{T}_{2^n}$ and a node $w_{i,j}$ represents a pair of its layer and its index: (i,j) for $i \in [0,n]$ and $j \in [0, 2^{n-i} - 1]$. We identify a value $x \in \mathbb{Z}_p$ with a node $(0,x)$. Consider a range $R = [u,v]$ for $0 \leq u \leq v \leq 2^n - 1$. For any range R, a node $w_{i,j} \in \mathbb{S}_{2^n}$ is called a cover node of R if all the descendant leaves of $w_{i,j}$ are in R. We write the set of such nodes as $\mathsf{cover}(R)$. For $w_{i,j} \in \mathbb{S}_{2^n}$ with $(i,j) \neq (n,0)$, let $\mathsf{parent}(w_{i,j})$ be the parent node of $w_{i,j}$. The range and point encodings are then defined as follows. For a range $R = [u,v]$ with $0 \leq u \leq v \leq 2^n - 1$, we let

$$\mathsf{rangeEnc}(R) := \{(i,a_i) \in \mathbb{S}_{2^n} \mid (i,a_i) \in \mathsf{cover}(R), \mathsf{parent}(i,a_i) \notin \mathsf{cover}(R)\}.$$

For a point $x \in [0, p-1]$, we let $\mathsf{pointEnc}(x)$ be the set of all ancestors of a node $(0,x)$ in $\mathbb{T}_{2^n}$ including the node $(0,x)$ itself. An example for point and range encoding is illustrated in Figs. 1 and 2, respectively. The main property is as follows: if for any range R, and any point $x \in [0, p-1]$, we have $|\mathsf{rangeEnc}(R) \cap \mathsf{pointEnc}(x)|$ equals to 1 if $x \in R$, and equals to 0 if $x \notin R$.

Note that we set the range R to reach the furthest to the right, *i.e.*, $v = 2^n - 1$, in our setting, which leads that $\mathsf{rangeEnc}(R)$ has no more than one node in each layer.

Our Protocol for $\mathsf{PlainLessThan}$**.** We show our protocol for $\mathsf{PlainLessThan}$ in Algorithm 4.

Correctness/Security. Suppose $x < y$. Hence, we have $y \in R = [x+1, 2^n - 1]$. Therefore, by the property of encodings, we have that exactly one element in $\mathsf{pointEnc}(y)$ equals to an element in $\mathsf{rangeEnc}(y)$. To perform OR over these equality tests for each layer, we simply sum their results up (as in Step 6). Note that, similarly to [2], Step 3 is for creating a dummy for all layers that are not contained in the range encoding of R. We can use the unused value $2^n - 1$ since $x, y < 2^n - 1$. Consequently, the equality tests corresponding to these layers always return false; we need them so that the number of layers to perform equality test will always be the same, namely, $n + 1$ layers (so as to ensure that there will be no additional information on the range R). As for security, it holds straightforwardly since we only call $\mathsf{PlainEqual}$ as a subroutine protocol (and note that $\mathsf{rangeEnc}$ and $\mathsf{pointEnc}$ are local algorithms).

3.5 WrapAround

We now describe a functionality that represents whether a reconstructed share wrap-arounds p. Its definition is as follows:

$$[\![y]\!] \leftarrow \mathsf{WrapAround}([\![x]\!]_1, [\![x]\!]_2),$$

where $y = 1$ if $[\![x]\!]_1 + [\![x]\!]_2 \geq p$ computed over $\mathbb{Z}$, and $y = 0$ otherwise.

Intuition/Approach. In this protocol, P_1 inputs $[\![x]\!]_1 \in \{0, 1, \ldots, p-1\}$, while P_2 inputs $[\![x]\!]_2 \in \{0, 1, \ldots, p-1\}$, and the output is a share $[\![y]\!]$ where

$$y = \begin{cases} 0 & \text{if } [\![x]\!]_1 + [\![x]\!]_2 = x \\ 1 & \text{if } [\![x]\!]_1 + [\![x]\!]_2 = x + p \end{cases}$$

where the sum is over the integers (*i.e.,* not modulo p).

Our Protocol for $\mathsf{WrapAround}$**.** A protocol for the wrap-around functionality can be done by simply computing $[\![y]\!] \leftarrow 1 - \mathsf{PlainLessThan}([\![x]\!]_1, p - [\![x]\!]_2)$.

Correctness/Security. The correctness holds since $y = 1 \iff [\![x]\!]_1 \geq p - [\![x]\!]_2$. As for security, it holds straightforwardly since we only call $\mathsf{PlainLessThan}$ as a subroutine.

3.6 Least Significant Bit

We describe the least significant bit functionality. This is defined by $[\![(x)_0]\!] \leftarrow \mathsf{LSB}([\![x]\!])$ where $(x)_0 := x \bmod 2$ is the LSB of x.

Intuition/Approach. The least significant bit can be evaluated by using LSB of shares and a flag representing a bit flip.

Our Protocol for LSB**.** Algorithm 5 presents our LSB protocol.

Algorithm 5. LSB Protocol

Functionality: $[\![(x)_0]\!] \leftarrow \mathsf{LSB}([\![x]\!])$
Input: Arithmetic shared value $[\![x]\!]$ over $\mathbb{F}_p$.
Output: $[\![(x)_0]\!]$ over $\mathbb{F}_p$ where $(x)_0 = x \bmod 2$.
1: P_1 locally extracts $b_1 := [\![x]\!]_1 \bmod 2$ and shares $[\![b_1]\!]$. At the same time, P_2 locally computes $b_2 := [\![x]\!]_2 \bmod 2$ and shares $[\![b_2]\!]$.
2: Compute $[\![w]\!] \leftarrow \mathbf{XOR}([\![b_1]\!], [\![b_2]\!]) = [\![b_1]\!] + [\![b_2]\!] - 2[\![b_1]\!] \cdot [\![b_2]\!]$.
3: Compute $[\![v]\!] \leftarrow \mathsf{WrapAround}([\![x]\!]_1, [\![x]\!]_2)$.
4: Output $[\![t]\!] \leftarrow \mathbf{XOR}([\![w]\!], [\![v]\!]) = [\![w]\!] + [\![v]\!] - 2[\![w]\!] \cdot [\![v]\!]$.

Correctness/Security. We can verify that

$$(x)_0 = \begin{cases} ([\![x]\!]_1)_0 \oplus ([\![x]\!]_2)_0 & \text{if } [\![x]\!]_1 + [\![x]\!]_2 = x \\ ([\![x]\!]_1)_0 \oplus ([\![x]\!]_2)_0 \oplus 1 & \text{if } [\![x]\!]_1 + [\![x]\!]_2 = x + p \end{cases}$$
$$= ([\![x]\!]_1)_0 \oplus ([\![x]\!]_2)_0 \oplus \mathsf{WrapAround}([\![x]\!]_1, [\![x]\!]_2).$$

As for security, it holds straightforwardly since we only call WrapAround and **XOR** as subroutines.

Note that this protocol can be run in 4 rounds. Step 1–2 takes 2 rounds, and can be run in parallel with Step 3 (3 rounds). Step 4 takes 1 round. Thus, it is 4 rounds in total, and its total communication is $4n^2 + 5$.

3.7 HalfTest

We describe a functionality that checks if the input is less than half of p as in [15]. This is defined by $[\![z]\!] \leftarrow \mathbf{HalfTest}([\![x]\!])$ where $z = (x < \frac{p}{2})$.

Our Protocol for HalfTest**.** As in Nishide-Ohta [15], this can be done by

$$[\![z]\!] \leftarrow 1 - \mathsf{LSB}([\![2x]\!]). \tag{4}$$

Correctness/Security. Security holds straightforwardly since we only call LSB as a subroutine.

3.8 Less-Than Comparison

Finally, we describe our less-than comparison functionality and a protocol for it. This is defined by $[\![z]\!] \leftarrow \mathbf{LessThan}([\![x]\!], [\![y]\!])$ where $z = (x < y)$.

Intuition/Approach. As shown in [15], we construct the LessThan protocol using HalfTest as a subroutine.

Our Protocol for LessThan**.** As shown in Nishide-Ohta [15], when we set $h_x := (x < \frac{p}{2})$, $h_y := (y < \frac{p}{2})$ and $h := (x - y \bmod p < \frac{p}{2})$, the required output can be computed as in the following equality relation:

$$z = h_x(1 - h_y) + (1 - h_x)(1 - h_y)(1 - h) + h_x h_y (1 - h). \tag{5}$$

For formality, we capture this protocol in Algorithm 6.

Algorithm 6. LessThan Protocol

Functionality: $[\![z]\!] \leftarrow \mathsf{LessThan}([\![x]\!], [\![y]\!])$
Input: Arithmetic shared values $[\![x]\!]$ and $[\![y]\!]$ over $\mathbb{F}_p$.
Output: Arithmetic shared value $[\![z]\!]$ over $\mathbb{F}_p$ where $z = (x < y)$.
1: Compute $[\![h_x]\!] \leftarrow \mathsf{HalfTest}([\![x]\!])$.
2: Compute $[\![h_y]\!] \leftarrow \mathsf{HalfTest}([\![y]\!])$.
3: Compute $[\![h]\!] \leftarrow \mathsf{HalfTest}([\![x - y]\!])$.
4: Compute $[\![z]\!] \leftarrow [\![h_x]\!] \cdot (1 - [\![h_y]\!]) + (1 - [\![h_x]\!]) \cdot (1 - [\![h_y]\!]) \cdot (1 - [\![h]\!]) + [\![h_x]\!] \cdot [\![h_y]\!] \cdot (1 - [\![h]\!])$.
5: Return $[\![z]\!]$.

Correctness/Security. As for security, it holds straightforwardly since we only call HalfTest as a subroutine.

4 Theoretical Efficiency

The efficiency of our protocol is measured in two aspects: round complexity and total communications. In literature, the round complexity is examined by the chain of multiplication protocols, and the total communication is examined by total invocations of multiplication protocols. However, any procedure that needs communication with other parties are crucial for execution time. Therefore, we count any communication such as "reveal", "send", or "share" as one round, which have been ignored in previous work. Thus, our rigid measurement counts up more rounds than that of previous work. In Table 1, we analyze previous constant-rounds secure comparison protocols from Damgård *et al.* [7] and Nishide-Ohta [15]. Moreover, we reconsider their protocols in the client-aided model and show that the rounds will be fewer in the model. We also show the result of our 6 rounds secure comparison protocol and its reduced round version.

Damgård *et al.*'s original secure comparison protocol [7] needs 79 rounds and total communication of $272n \log n + 138n + 22 \log n + 24(\log n)^2 + 24n \log \log n + 12$ field elements, since the protocol consists of two Bits protocols (69 rounds and $136n \log n + 56n + 8 \log n + 12(\log n)^2 + 12n \log \log n + 6$ total communications per Bits) and Bit-LT protocol (13 rounds including 3 rounds for random generation and $26n + 6 \log n$ total communications). Similarly, Nishide-Ohta's original secure comparison protocol [15] needs 28 rounds and total communication of $168n + 36 \log n + 16$, since the protocol consists of 3 LSB protocols (in parallel) and 2 MULT protocols, where LSB is 26 rounds and $32n + 40 \log n$ total communications and MULT is 1 round and 2 total communications.

In the client-aided model, we can omit procedures of generating randomness so that Bits will be 60 rounds and total communications will be $52n \log n + 16n + 5 \log n + 10(\log n)^2 + 10n \log \log n$, while Bit-LT will be 10 rounds and $12n + 5 \log n$ total communications. This let Damgård *et al.*'s protocol be 70 rounds and $144n \log n + 52n + 64 \log n + 64(\log n)^2 + 64n \log \log n - 6$ total communications. Similarly, LSB will be 12 rounds and $12n + 5 \log n + 3$ total communications in the client-aided model, which makes Nishide-Ohta's

Table 2. Execution times of protocols

Operations	Round	Offline [ms]	Online comp. [us]	Estimated online comm. [ms]	Estimated online total [ms]
ADD	0	–	0.000,10	–	–
$\mathsf{MULT}_{\mathbf{pub}}$	0	–	0.000,20	–	–
$\mathsf{MULT}_{\mathbf{priv}}$	1	0.001,355	0.000,10	72	72
MULT^* ($\ell = 32$)	2	0.059,558	0.681,70	144	144
Power ($\ell = 32$)	2	0.059,558	0.667,20	144	144
AND^* ($m = 32$)	2	0.059,558	1.044	144	144
PlainEqual	3	0.100,386	2.461	216	216
PlainLessThan	3	1.823	63.783	216	216
WrapAround	3	1.823	63.416	216	216
LSB	4	1.821	76.028	288	288
HalfTest	4	1.821	76.424	288	288
LessThan	6	5.574	273.600	432	432

secure comparison protocol 14 rounds and $36n + 15 \log n + 13$ total communications.

Our secure comparison protocol LessThan consists of three HalfTest (more precisely, LSB that is 4 rounds and $4n^2 + 5$ total communications as explained in Sect. 3.6) protocols and a degree-3 polynomial (more precisely, 5 MULT, which has 10 total communications). Naïvely computing the polynomial (without merging the same computations), our protocol has 6 rounds and at most $12n^2 + 25$ total communications.

Further Reducing Rounds. We can combine the step 4 of the LSB protocol and Eqs. (4) to (5) to a degree-6 polynomial. Since each variable is 0 or 1 shared in $\mathbb{Z}_p$, multiplications can be done by AND^* protocol in 2 rounds. This results in a LessThan protocol with 5 rounds. The maximum transmitting data amount in one round is within the limitation assumed in our WAN setting. See Appendix A for more details.

5 Experimental Efficiency

In this section, we give performance evaluation of our secure comparison protocol LessThan based on our experiments. For the evaluation, we implement the protocol in C++ programming language using a desktop PC (Xeon E5-2699 v4, 2.20 GHz), Linux Ubuntu 16.04.3 LTS, and a compiler GCC version 5.4.0. Throughout the experiments, we set the prime number $p = 4294967291 = 2^{32} - 5$. Note that our proposed algorithms are independent from the choice of the prime number. Also note that the architecture we used supports 64 bits instruction

set, and as shown above, the bit length of p is 32; thus it is unnecessary to use multi-precision arithmetic. To implement our protocol, we do not use assembler, any optimization technique by hand, and any optimized arithmetic software library. We use a special file "`/dev/urandom`" to implement cryptographically secure pseudorandom number generator, and optimizations by the compiler with an option "`-O3`." From this implementation, we evaluate computational time of our protocols, the number of communication rounds, and communication sizes. Based on them, we further estimate the total execution time assuming that the two servers are connected via Wide Area Network (WAN) whose bandwidth and network delay are the same as those in [14] (Namely, we set the bandwidth to be 9 MB/s and the network delay to be 72 ms).

Table 2 shows the execution times of our LessThan protocol and its subroutines. The column of "Offline" represents time for a client to generate multiplication triples and correlated randomness, and the column "Online Comp." represents the computation time of each protocol without communication. The column "Estimated Online Comm." represents the estimated communication time of each protocol by using the assumption described above. Namely, it takes 72 ms per a round. The column "Estimated Online Total" represents the estimated total execution time which is the sum of "Online Comp." and "Estimated Online Comm". For taking the execution times, we set the numbers of inputs of protocols MULT*, Power, and AND* as 32 (*i.e.,* in Algorithm 1, $\ell = 32$, and in Algorithm 2, $m = 32$). We note that in the total execution time, network delay is the dominant factor, and compared to this, influence of computational time and communication size is almost ignorable. Therefore, it is important that the number of communication rounds should be reduced as much as possible when combining a secure comparison protocol to construct concrete applications. For reducing the round complexity, our proposed algorithms can be adopted to the vectorization (*i.e.,* operating on vectors) same as in [14] and batch execution techniques.

Acknowledgement. This work was supported by JST CREST JPMJCR1688.

A Further Round-Reduced LessThan Protocol

As we mentioned in Sect. 4, our LessThan protocol can be executed in 5 rounds as follows: We can combine the step 4 of the LSB protocol and Eqs. (4) to (5) to a degree-6 polynomial. In particular, this technique breaks down our LessThan to three LSBish protocols (3 rounds and $4n^2 + 3$ total communications) and a degree-6 polynomial F defined below: $F = w+v-w_xw-w_xv-wv_x-v_xv+w_y-w_yw-w_yv-w_xw_y-w_yv_x+v_y-wv_y-v_yv-w_xv_y-v_xv_y+2(-wv+w_xwv+w_ywv+w_xw_yw+w_xw_yv+w_ywv_x+w_yv_xv+wv_xv+w_xwv_x+w_xv_xv+w_xw_yv_x+wv_yv+w_xwv_y+w_xv_yv+wv_xv_y+v_xv_yv+w_xv_xv_y-w_yv_y+w_ywv_y+w_yv_yv+w_xw_yv_y+w_yv_xv_y)+4(-w_xwv_xv-w_xw_ywv-w_ywv_xv-w_xw_ywv_x-w_xw_yv_xv-wv_xv_yv-w_xwv_xv_y-w_xv_xv_yv-w_ywv_yv-w_xw_ywv_y-w_yv_xv_yv-w_ywv_xv_y-w_xw_yv_yv-w_xw_yv_xv_y-v_yw_xwv)+8(w_xw_ywv_xv+w_xwv_xv_yv+w_xw_ywv_yv+w_ywv_xv_yv+w_xw_ywv_xv_y+w_xw_yv_xv_yv)-16w_xw_ywv_xv_yv.$

The function F contains 13 degree-2, 26 degree-3, 15 degree-4, 6 degree-5, and 1 degree-6 terms, which can be computed in 2 rounds and 292 total communications. Since each variable is 0 or 1 shared in $\mathbb{Z}_p$, multiplications of the function F can be done by AND^* protocol in 2 rounds. This results in a LessThan protocol with 5 rounds and the total communication is $3(4n^2 + 3) + 292 = 12n^2 + 301$.

Algorithm 7. LSBish Protocol

Functionality: $(\llbracket w \rrbracket, \llbracket v \rrbracket) \leftarrow \mathbf{LSBish}(\llbracket x \rrbracket)$
Input: Arithmetic shared value $\llbracket x \rrbracket$ over $\mathbb{F}_p$.
Output: $\llbracket (x)_0 \rrbracket$ over $\mathbb{F}_p$ where $(x)_0 = x \bmod 2$.
1: P_1 locally extracts $b_1 := \llbracket x \rrbracket_1 \bmod 2$ and shares $\llbracket b_1 \rrbracket$. At the same time, P_2 locally computes $b_2 := \llbracket x \rrbracket_2 \bmod 2$ and shares $\llbracket b_2 \rrbracket$.
2: Compute $\llbracket w \rrbracket \leftarrow \mathbf{XOR}(\llbracket b_1 \rrbracket, \llbracket b_2 \rrbracket) = \llbracket b_1 \rrbracket + \llbracket b_2 \rrbracket - 2\llbracket b_1 \rrbracket \llbracket b_2 \rrbracket$.
3: Compute $\llbracket v \rrbracket \leftarrow \mathsf{WrapAround}(\llbracket x \rrbracket_1, \llbracket x \rrbracket_2)$.
4: Output $(\llbracket w \rrbracket, \llbracket v \rrbracket)$.

Algorithm 8. LessThan Protocol (described explicitly)

Functionality: $\llbracket z \rrbracket \leftarrow \mathsf{LessThan}(\llbracket x \rrbracket, \llbracket y \rrbracket)$
Input: Arithmetic shared values $\llbracket x \rrbracket$ and $\llbracket y \rrbracket$ over $\mathbb{F}_p$.
Output: Arithmetic shared value $\llbracket z \rrbracket$ over $\mathbb{F}_p$ where $z = (x < y)$.
1: Compute $(\llbracket w_x \rrbracket, \llbracket v_x \rrbracket) \leftarrow \mathbf{LSBish}(\llbracket x \rrbracket)$.
2: Compute $(\llbracket w_y \rrbracket, \llbracket v_y \rrbracket) \leftarrow \mathbf{LSBish}(\llbracket y \rrbracket)$.
3: Compute $(\llbracket w \rrbracket, \llbracket v \rrbracket) \leftarrow \mathbf{LSBish}(\llbracket x - y \rrbracket)$.
4: Compute F by using **AND**, addition, and multiplication with public value.
5: Return $\llbracket z \rrbracket$.

B Round Complexity and Communication Complexity

In Table 3, we put round complexity and total communications of each protocol from [7,15]. In Table 4, we show round complexity and total communication of our LessThan protocol and its subroutines. These are used for calculating rounds and total communication in Table 1. We note that a more detailed value for total communication of the Damgård et al. [7] protocol is $176n \log n + 70n + 84 \log n + 80(\log n)^2 + 80n \log \log n - 6$ (which is reduced to $144n \log n + 52n + 64 \log n + 64(\log n)^2 + 64n \log \log n - 6$ for the client-aided version).

Table 3. Number of rounds and total communication of each protocol from [7,15]

Protocol	Round	Total comm. (elements)
Unbounded-fan-in OR with ℓ inputs	5	$10\ell - 2$
Prefix OR	11	$10n + 20\log n - 2$
RAN_2	3	4
SOLVED-BITS	14	$18n + 20\log n - 1$
BIT-LT	13	$14n + 20\log x - 2$
BIT-ADD	25	$44n\log n - 2n - 4\log n + 20(\log n)^2 + 20n\log\log n$
CARRIES	25	$44n\log n - 2n - 4\log n + 20(\log n)^2 + 20n\log\log n$
$\text{PRE}_\circ$	24	$44n\log n - 4n - 4\log n + 20(\log n)^2 + 20n\log\log n$
BITS	69	$88n\log n + 28n + 32\log n + 40(\log n)^2 + 40n\log\log n - 2$
LessThan of [7]	79	$176n\log n + 70n + 84\log n + 80(\log n)^2 + 80n\log\log n - 6$
LSB	26	$32n + 40\log n$
LessThan of [15]	28	$96n + 120\log n + 4$

Table 4. Number of rounds and total communication of our LessThan protocol and its subroutines

Protocol	Round	Total comm. (elements)
$\mathsf{MULT}_{\mathbf{priv}}$	1	2
MULT*, Power, AND* (ℓ inputs)	2	2ℓ
PlainEqual	3	$4n$
PlainLessThan, WrapAround	3	$4n^2$
LSB, HalfTest	4	$4n^2 + 5$
LessThan (Implemented in Sect. 5)	6	$12n^2 + 25$
LessThan (1 round reduced)	5	$12n^2 + 301$

One might wonder if the amount of transmitting field elements during any round exceeds the limitation, *i.e.,* $9\,\text{MB/s} \times 72\,\text{ms} = 648\,\text{KB}$. If the amount of transmitting data (elements of $\mathbb{F}_p$) exceeded the limitation, the protocol would need extra rounds to send all the data. In our LessThan protocol, a larger amount of data is needed during executing PlainLessThan protocol, more specifically n PlainEqual protocols in parallel. This protocol sends at most $2n^2$ field elements at one round. This leads to that our LessThan sends at most $6n^2$ field elements, since it run three HalfTest (constructed by WrapAround that has PlainLessThan as subroutine) protocols at once. When p is 32-bit prime, *i.e.,* $n = 32$, our LessThan protocol sends at most $6144 = 6 \times 32^2$ field elements (196608 bits) in one round, which is less than the limitation; 648 KB.

References

1. 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, pp. 805–817 (2016)
2. Attrapadung, N., Hanaoka, G., Kiyomoto, S., Mimoto, T., Schuldt, J.C.N.: A taxonomy of secure two-party comparison protocols and efficient constructions. In: 15th Annual Conference on Privacy, Security and Trust, PST 2017, Calgary, Canada, 28–30 August 2017. IEEE (2017)
3. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 420–432. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-46766-1_34
4. Blake, I.F., Kolesnikov, V.: Strong conditional oblivious transfer and computing on intervals. In: Lee, P.J. (ed.) ASIACRYPT 2004. LNCS, vol. 3329, pp. 515–529. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30539-2_36
5. Blake, I.F., Kolesnikov, V.: Conditional encrypted mapping and comparing encrypted numbers. In: Di Crescenzo, G., Rubin, A. (eds.) FC 2006. LNCS, vol. 4107, pp. 206–220. Springer, Heidelberg (2006). https://doi.org/10.1007/11889663_18
6. Catrina, O., de Hoogh, S.: Improved primitives for secure multiparty integer computation. In: Garay, J.A., De Prisco, R. (eds.) SCN 2010. LNCS, vol. 6280, pp. 182–199. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15317-4_13
7. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 285–304. Springer, Heidelberg (2006). https://doi.org/10.1007/11681878_15
8. Damgård, I., Geisler, M., Krøigaard, M.: Homomorphic encryption and secure comparison. IJACT **1**(1), 22–31 (2008)
9. Damgard, I., Geisler, M., Kroigard, M.: A correction to 'efficient and secure comparison for on-line auctions'. Int. J. Appl. Cryptogr. **1**(4), 323–324 (2009)
10. David, B., Dowsley, R., Katti, R., Nascimento, A.C.A.: Efficient unconditionally secure comparison and privacy preserving machine learning classification protocols. In: Au, M.-H., Miyaji, A. (eds.) ProvSec 2015. LNCS, vol. 9451, pp. 354–367. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26059-4_20
11. Garay, J., Schoenmakers, B., Villegas, J.: Practical and secure solutions for integer comparison. In: Okamoto, T., Wang, X. (eds.) PKC 2007. LNCS, vol. 4450, pp. 330–342. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71677-8_22
12. Goldreich, O.: The Foundations of Cryptography - Volume 2, Basic Applications. Cambridge University Press, Cambridge (2004)
13. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: 1987 Proceedings of the 19th Annual ACM Symposium on Theory of Computing, pp. 218–229 (1987)
14. Mohassel, P., Zhang, Y.: SecureML: a system for scalable privacy-preserving machine learning. In: SP 2017, pp. 19–38 (2017)
15. Nishide, T., Ohta, K.: Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In: Okamoto, T., Wang, X. (eds.) PKC 2007. LNCS, vol. 4450, pp. 343–360. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71677-8_23

16. Reistad, T.I.: Multiparty comparison - an improved multiparty protocol for comparison of secret-shared values. In: SECRYPT 2009, pp. 325–330 (2009)
17. Reistad, T.I., Toft, T.: Secret sharing comparison by transformation and rotation. In: Desmedt, Y. (ed.) ICITS 2007. LNCS, vol. 4883, pp. 169–180. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10230-1_14
18. Reistad, T., Toft, T.: Linear, constant-rounds bit-decomposition. In: Lee, D., Hong, S. (eds.) ICISC 2009. LNCS, vol. 5984, pp. 245–257. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14423-3_17
19. Rivest, R.L.: Unconditionally secure commitment and oblivious transfer schemes using private channels and a trusted initializer (1999, unpublished manuscript)
20. Schneider, T., Zohner, M.: GMW vs. yao? Efficient secure two-party computation with low depth circuits. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 275–292. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39884-1_23
21. Schoenmakers, B., Tuyls, P.: Practical two-party computation based on the conditional gate. In: Lee, P.J. (ed.) ASIACRYPT 2004. LNCS, vol. 3329, pp. 119–136. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30539-2_10
22. Veugen, T.: Encrypted integer division and secure comparison. Int. J. Appl. Cryptol. **3**(2), 166–180 (2014)
23. Veugen, T., Blom, F., de Hoogh, S.J.A., Erkin, Z.: Secure comparison protocols in the semi-honest model. J. Sel. Top. Sig. Process. **9**(7), 1217–1228 (2015)
24. 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. IEEE Computer Society (1986)