Oblivious Linear Group Actions and Applications

Nuttapong Attrapadung
AIST, Japan
n.attrapadung@aist.go.jp

Goichiro Hanaoaka
AIST, Japan
hanaoka-goichiro@aist.go.jp

Takahiro Matsuda
AIST, Japan
t-matsuda@aist.go.jp

Hiraku Morita
University of St. Gallen, Switzerland
hiraku.morita@unisg.ch

Kazuma Ohara
AIST, Japan
ohara.kazuma@aist.go.jp

Jacob C. N. Schuldt
AIST, Japan
jacob.schuldt@aist.go.jp

Tadanori Teruya
AIST, Japan
tadanori.teruya@aist.go.jp

Kazunari Tozawa
University of Tokyo, Japan
tozawaka@edu.k.u-tokyo.ac.jp

ABSTRACT

In this paper we propose efficient two-party protocols for obliviously applying a (possibly random) linear group action to a data set. Our protocols capture various applications such as oblivious shuffles, circular shifts, matrix multiplications, to name just a few. A notable feature enjoyed by our protocols, is that they admit a round-optimal (more precisely, one-round) online computation phase, once an input-independent off-line computation phase has been completed. Our oblivious shuffle is the first to achieve a round-optimal online phase. The most efficient instantiations of our protocols are obtained in the so-called client-aided client-server setting, where the offline phase is run by a semi-honest input party (client) who will then distribute the generated correlated randomness to the computing parties (servers). When comparing the total running time to the previous best two-party oblivious shuffle protocol by Chase et al. (Asiacrypt 2020), our shuffle protocol in this client-aided setting is up to 105 times and 152 times faster, in the LAN and WAN setting, respectively. We additionally show how the Chase et al. protocol (which is a standard two-party protocol) can be modified to leverage the advantages of the client-aided setting, but show that, even doing so, our scheme is still two times faster in the online phase and 1.34 times faster in total on average.

An additional feature of our protocols is that they allow to re-invoke a previously generated group action, or its inverse, in subsequent runs. This allows us to utilize randomize-then-reveal techniques, which are crucial for constructing efficient protocols in complex applications. As an application, we construct a new oblivious sorting protocol implementing radix sort. Our protocol is based on a similar approach to the three-party protocol by Chida et al. (IACR ePrint 2019/965), but using our oblivious shuffle as a building block as well as various optimizations, we obtain a two-party protocol (in the client-aided setting) with improved online running time and a reduced number of rounds. As other applications, we also obtain efficient protocols for oblivious selection, oblivious unit-vectorization, oblivious multiplexer, oblivious polynomial evaluation, arithmetic-to-boolean share conversions, and more.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11.
https://doi.org/10.1145/3460120.3484584

CCS CONCEPTS

• **Theory of computation → Cryptographic protocols**.

KEYWORDS

secure computation; secret sharing; oblivious shuffle; oblivious sorting

ACM Reference Format:
Nuttapong Attrapadung, Goichiro Hanaoaka, Takahiro Matsuda, Hiraku Morita, Kazuma Ohara, Jacob C. N. Schuldt, Tadanori Teruya, and Kazunari Tozawa. 2021. Oblivious Linear Group Actions and Applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3460120.3484584

1 INTRODUCTION

Secure two-party and multi-party computation (2PC/MPC) allows mutually distrusting parties to jointly evaluate a function on their private inputs without revealing anything beyond the output of the function. Secure computation has come a long way from the classic general feasibility results in the eighties [2, 16, 46]. In this paper, we study secret-sharing based secure two-party computation, which is one of the well-studied branches of MPC due to its light computation and its applicability to complex applications, such as privacy-preserving machine learning (*e.g.,* [32]). We aim to propose efficient protocols for securely computing a large class of functions related to *group actions*. The notion of a group acting on a set is a general and important one which links abstract algebra to wide ranges of branches in mathematics, such as geometry, linear algebra, differential equations, to name just a few [15].

Our Focus: Oblivious Linear Group Actions. In this paper, we propose what we call *oblivious linear group actions*. A protocol for oblivious linear group actions allows two parties to jointly apply a random "group action" to data (a set), and obtain secret shares of the result – without any party learning the action. One of the main applications of oblivious linear group actions is an "oblivious shuffle" (also called a "secret-shared shuffle" in [9]), which is itself a fundamental building block to many protocols, such as secure

function evaluation over private set intersection [12, 38, 39], secure database join [27, 30], oblivious sorting [4, 11].

Another notable application of linear group action is General Linear Group Action, *i.e.,* matrix multiplications, which are fundamental in secure arithmetic computations.

Offline/Online Paradigm. Towards constructing efficient secure computation protocols, it is instructive to consider a protocol as being composed of two phases: a pre-processing (or offline) phase and an online phase. The pre-processing phase occurs before inputs (the data from parties) are determined and hence can be executed at anytime beforehand; it produces as output "correlated-randomness" which will be consumed later in the online phase. Then, the online phase will make use of the correlated randomness in conjunction with the parties' intended inputs, and will produce the output of the protocol. This so-called offline/online paradigm is considered in many works, and is rooted in the celebrated Beaver triple based multiplication protocols [1]. Correlated randomness can be generated in many ways. It can be done via protocols run either by the same parties who execute the online phase or a set of independent servers (i.e. the so-called server-aided settings [23, 41]). Alternatively, it can be done very efficiently via a semi-honest third party. The latter approach works particularly well in the *client-server setting*, where a set of clients provide the inputs and a set of servers perform the computation. The aforementioned third-party can then be a semi-honest client who assists the servers by generating all the required correlated randomness in the offline phase. This is called *client-aided client-server setting* [32, 33, 40].

Our Goal: Fast Protocols with Efficient Online Phase. Our goal is to construct fast protocols for oblivious linear group actions. First and foremost, we aim to construct *round-optimal* online protocols, namely, *one-round* protocols. A one-round protocol has to be designed so that the parties can send messages *simultaneously* (*i.e.,* in parallel) to each other without having to wait for the other parties to process local data or incoming messages. This simultaneous nature admits fast protocol execution and is hence desirable. For the offline phase, our main focus will be on the client-aided client-server approach, which allows very efficient instantiations.

Our Main Contributions. We propose a simple two-party oblivious linear group action protocol that admits a one-round online phase (and is thus online round-optimal). Let G be a group and V be a set and $\psi : V \times G \to V$ be a (linear) group action. An oblivious linear group action protocol lets two parties input random shares of $x \in V$ and output random shares of $\psi(x, r)$ for random $r \in G$, without any party learning r. Our protocol is secure against static semi-honest adversaries with one corruption.

Our protocol allows parties to store the shares, denoted by $\langle r \rangle$, of a previously generated element r (still, without learning r) so that they can "re-invoke" the group action but on another input $y \in V$ to obtain $\psi(y, r)$ (in the shared form).[1] Moreover, we can generate $\langle r' \rangle$ from $\langle r \rangle$, *non-interactively* for some r' that depends on r. In particular, when setting r' to be r^{-1}, we obtain an 'inverse" group action protocol. The non-interactive conversion is also applicable to some compositions of multiple group elements. This allows us to reduce the communication complexity of protocols using multiple linear group actions; we demonstrate this in various applications (e.g. oblivious sorting described below).

Our Applications. We obtain many applications as follows.

- *Direct Applications.* By instantiating groups and their actions, we obtain direct applications including online-round-optimal (one-round) protocols for the following.
  - Oblivious shuffle, when considering G to be a group of permutations. This is the first shuffle protocol with a one-round online protocol. For the total running time (combined offline and online time), our shuffle protocol in the client-aided two-party setting is up to around 152 times faster than the unmodified Chase *et al.* [9] protocol (which is a standard two-party protocol not taking advantage of client computation) in the WAN setting, and up to around 105 times faster in the LAN setting, when shuffling 2^{32}-length vectors. We additionally modify the Chase et al. protocol to leverage the client-aided setting and compare the resulting protocol to ours, see Section 1.1.
  - Oblivious circular shift, when considering G to be a cyclic group. This is a special case of, and is more efficient than, shuffles. It has nice applications in computation related to positions in vectors, such as what we call Oblivious selection (the shared-input/output variant of a 1-out-of-n oblivious transfer) and Oblivious unit-vector conversion (securely converting an integer encoding to its "one-hot" encoding).[2] We also obtain a new one-round online protocol for arithmetic-to-boolean (A2B) share conversions (for polynomial-size domains).
  - Oblivious matrix-vector/matrix-matrix multiplication, when considering G to be the general linear group (i.e. the group of invertible matrices). Thanks to the "reusable" action feature, we obtain an efficient "batch" protocol for secure matrix-vector multiplications with the same matrix and varied k vectors. This reduces the communication cost from $2kn^2$ of the naive approach to $2n^2 + kn$, where n is the vector dimension.
- *A More Complex Application: Oblivious Sorting.* Thanks to our feature of "re-invocability", we are able to apply our oblivious shuffles with *composed* permutations to the radix sort protocol of Chida *et al.* [11], resulting in a new efficient sort protocol. Our protocol in the client-aided two-party setting is up to 2.57 times faster in the online phase than the protocol of [11], which is a standard three-party protocol. Moreover, our protocol has a lower number of online rounds compared to that of [11].

Further Results. We also consider more instances of group actions such as XOR permutations, polynomial translations, and actions by direct product groups. Instantiating these, we obtain *e.g.,* oblivious XOR-shuffle. From these, we further obtain more applications such as oblivious multiplexer, oblivious polynomial evaluation (in the secret-shared form), secure (multi-input) multiplication. All of these feature one-round online protocols; in particular, to the best of our knowledge, our protocols for multiplexer and polynomial evaluation are the first explicit secret-sharing-based protocols (for their respective computations) that achieve the online-one-round feature. Due to limited space, we defer them to Appendix C and D.

[1] Note that each new invocation requires fresh correlated randomness, in the same manner as the Beaver multiplication protocol [1] consumes fresh Beaver triples.

[2] The one-hot encoding of an integer i refers to the unit-vector with 1 in only the i-th position (and others are all 0). It has been widely used for encoding categories in the (privacy-preserving) machine-learning literature [10, 42].

While our main focus is on the client-aided setting, for completeness, we also construct two-party protocols for the offline phase that allows the two computing parties to execute our protocol without assistance from a client. Due to limited space, we defer the details of these protocols to the full version. In particular, our sort protocol in combination with our pre-processing protocols yields a standard two-party protocol. This provides another improvement over the Chida *et al.* [11] protocol in terms of supported adversarial corruption (from honest majority to dishonest majority).

1.1 Technical Overview

Difficulties towards One-round Shuffle Protocol. To see an inherent difficulty of obtaining one-round protocols, we will look into oblivious shuffle, which is a special case of oblivious linear group actions. Let R be a ring and n be a positive integer. Consider data as a vector $\boldsymbol{x} \in R^n$. We use 2-out-of-2 secret sharing for $\boldsymbol{x}$, namely, $\boldsymbol{x} = \boldsymbol{x}_1 + \boldsymbol{x}_2$, for random $\boldsymbol{x}_1$, and denote $\boldsymbol{x}_i$ as shares for the party P_i for $i = 1, 2$. An oblivious shuffle protocol lets two parties input random shares of data $\boldsymbol{x}$ and outputs random shares of $\rho(\boldsymbol{x})$, the permuted $\boldsymbol{x}$ via a random permutation ρ, without any party learning ρ. To the best of our knowledge, previously known two-party oblivious shuffle protocols, including Chase *et al.* [9], are constructed by *sequentially* applying two instances of its "half" building block called Permute+Share protocol [9]. A Permute+Share protocol lets two parties input shares of $\boldsymbol{x}$ and one party, say P_1, also input a permutation of its choice ρ_1.[3] An oblivious shuffle protocol can then be constructed by first running the Permute+Share protocol with P_1 choosing ρ_1 and then another instance with P_2 choosing another permutation ρ_2. The resulting oblivious shuffle is achieved with ρ being composed as $\rho_2 \circ \rho_1$. The very nature of this paradigm that the two sub-permutations have to be applied *sequentially* makes the resulting protocol require at least 2 rounds at best, and more precisely, about two times the round complexity of the Permute+Share (which is itself usually more than 1 round).

Our Approach: Correlated Double-Sharing. We explain our method to overcome the difficulties in the case of oblivious shuffle. We recall that the previous paradigm sequentially computes *one pair* of *independent* sub-permutations, each via a Permute+Share protocol. Our approach deviates from this by using *two pairs* of sub-permutations that are *dependent* in such a way that both pairs compose to *the same permutation*. More precisely, we set

$$\rho = \rho_1 \circ \tau_2 = \rho_2 \circ \tau_1,$$

so that P_1 knows only ρ_1, τ_1 and P_2 knows only ρ_2, τ_2. In particular, they both do not learn the full permutation ρ. We call this a "double-sharing" of ρ. Intuitively, this dependency allows each party P_i to apply the first sub-permutation in the composition, namely, τ_i, directly to its input share $\boldsymbol{x}_i$, *without having to wait* for the other party's action. This, together with our design that allows the second sub-permutation ρ_i to be applied *only locally*, results in a one-round protocol. To proceed with this design, we must carefully use the double-sharing such that permutations ρ_i, τ_i are not exposed to the other party, as this would expose the full ρ. We resolve this by "correlating" the double sharing via three random vectors $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in R^n$ in a "symmetric" manner. More precisely, our protocol works as follows. In the pre-processing phase, we let

$$P_1 \text{ hold } \rho_1, \tau_1, \boldsymbol{a}, \text{ and } \boldsymbol{\alpha} := \rho_1(\boldsymbol{b}) + \boldsymbol{c},$$
$$P_2 \text{ hold } \rho_2, \tau_2, \boldsymbol{b}, \text{ and } \boldsymbol{\beta} := \rho_2(\boldsymbol{a}) - \boldsymbol{c}.$$

Now, in the online phase, P_1 has a share $\boldsymbol{x}_1$ and P_2 has a share $\boldsymbol{x}_2$ as inputs. The protocol is done by simply letting

$$P_1 \text{ send } P_2\ \boldsymbol{v} := \tau_1(\boldsymbol{x}_1) + \boldsymbol{a},$$
$$P_2 \text{ send } P_1\ \boldsymbol{w} := \tau_2(\boldsymbol{x}_2) + \boldsymbol{b}.$$

Crucially, observe that these two messages can be communicated *simultaneously*! (and hence in one round). Finally, locally, P_1, P_2 sets its share as $\boldsymbol{y}_1, \boldsymbol{y}_2$, respectively, where

$$\boldsymbol{y}_1 := \rho_1(\boldsymbol{w}) - \boldsymbol{\alpha}, \qquad \boldsymbol{y}_2 := \rho_2(\boldsymbol{v}) - \boldsymbol{\beta}.$$

The protocol works correctly since

$$\boldsymbol{y}_1 = \rho_1(\tau_2(\boldsymbol{x}_2) + \boldsymbol{b}) - (\rho_1(\boldsymbol{b}) + \boldsymbol{c}) = \rho(\boldsymbol{x}_2) - \boldsymbol{c}$$
$$\boldsymbol{y}_2 = \rho_2(\tau_1(\boldsymbol{x}_1) + \boldsymbol{a}) - (\rho_2(\boldsymbol{a}) - \boldsymbol{c}) = \rho(\boldsymbol{x}_1) + \boldsymbol{c}$$

and hence $\boldsymbol{y}_1 + \boldsymbol{y}_2 = \rho(\boldsymbol{x}_2) + \rho(\boldsymbol{x}_1) = \rho(\boldsymbol{x})$, as required.

To generate the correlated randomness, we focus on the client-aided client-server setting, which requires the client to generate the above $\rho_i, \tau_i, \boldsymbol{a}, \boldsymbol{\alpha}, \boldsymbol{b}, \boldsymbol{\beta}$ and send this to the two respective parties (servers) in the offline phase.

Revisiting Chase *et al.*. Our approach deviates from the Chase *et al.* protocol (CGP) [9], which was not designed with the offline/online paradigm in mind or with consideration of a client initiating the computation and potentially aiding the computing parties. Specifically, the (implicit) correlated randomness used in the Chase *et al.* protocol is produced online directly (and implicitly) by the two computing parties. For completeness, in Appendix A, we modify their protocols to leverage the client-aided setting, and show that, even when doing so, our protocol is still two times faster in the online phase. Intuitively, this is because their protocol uses two independent sub-permutations, and hence inherently requires at least 2 rounds in the online phase, even when optimizing the protocol to take advantage of the client-aided setting. We also remark that, for the total running time, ours is 1.34 times faster on average.

1.2 Related Works

Related works on Oblivious Shuffles. To the best of our knowledge, there exist only few efficient concrete *two-party* shuffle protocols, with the state-of-the-art being that of Chase [9]. It is based on Permute+Share, which itself can be constructed in many ways [21, 31]; the most efficient Permute+Share, based on variants of oblivious transfer, is proposed in [9]. Viewing shuffles as applying (random permutation) matrix multiplications, one can use generic MPC, but this would yield much less efficient protocols (see [9]).

We study two-party shuffles (with a client-aided off-line phase). For *three-party* shuffles, more schemes have been proposed. Laur *et al.* [28] proposed three protocols, based on permutation matrices, sorting, and re-sharing; these require a non-constant number of rounds that grows with the number of elements n. Movahedi *et al.* [34] proposed a protocol with $O(\log n)$ rounds, but with slightly non-standard security. Chida *et al.* [11] proposed a constant-round protocol with a two-round online phase.

[3] Note that there are at least two variants of Permute+Share as per [9]. The one we described here is the one with shared inputs.

Related Works on Oblivious Sorting. There are basically two approaches to oblivious sorting. The first category is *data-dependent* approaches, such as oblivious quicksort [18]. Such schemes use oblivious shuffles beforehand so as to be able to reveal the comparison results between any two entries, which are then used in a data-dependent comparison-based sort. This enables the protocol to run fast in practice, but it leaks the number of equal entries. The second category is data-independent approaches, such as oblivious naive comparison sorting [3, 4], oblivious sorting networks [22, 44], and oblivious radix sort [3, 4, 11, 17, 26, 47]. These protocols are data-independent as their procedures have a-priori fixed structures. The oblivious naive comparison sorting protocol by [3, 4] uses oblivious shuffle protocols to enable revealing of (oblivious) comparison results (of all pairs in the vector). This revealing incurs the same problem as oblivious quicksort [18], namely, the number of equal entries is leaked. Sorting based on sorting networks do not use oblivious shuffles, but still require a large number of oblivious comparison protocol executions, which might result in large overall communication costs. Oblivious radix sort protocols do not require oblivious comparison protocols, and are hence expected to perform better (especially for large input vectors). The state-of-the-art *three-party* protocol for oblivious radix sort is that of Chida *et al.* [11]. To the best of our knowledge, prior to our work, there were no known *two-party* oblivious radix sort protocols that are comparable in efficiency to [11]. We refer a more survey on oblivious sort to [43].

Other Protocols. Laur *et al.* proposed a generic construction of protocols for oblivious selection and (implicitly) oblivious unit-vectorization [28]. Their protocols have $\tau_{eq} + \tau_{mul}$ and τ_{eq} rounds, respectively, where τ_{eq} and τ_{mul} are respectively the number of rounds required for equality check and multiplication. A2B conversions are considered in the widely used ABY/ABY3 framework [14, 29], who proposed logarithmic round protocols. We consider A2B for polynomial-size domains. The only available one-online-round A2B protocol is that of Boyle *et al.* [7], based on function secret-sharing, which is computationally secure. We propose an alternative construction that is information-theoretically secure, but requires larger correlated randomness.

2 PRELIMINARIES

Basic Notation. For a set X, $x \xleftarrow{\$} X$ denotes choosing an element x uniformly at random from X, and $|X|$ denotes the size of X. For any x, y, $(x \stackrel{?}{=} y)$ denotes the operation that returns 1 if $x = y$ and 0 otherwise. $\stackrel{?}{<}$ and $\stackrel{?}{\in}$ are defined similarly. For a natural number n, S_n denotes the set of all automorphisms (*i.e.,* permutations) on $\{1, \ldots, n\}$. For a vector $\boldsymbol{v}$, $\boldsymbol{v}_i$ denotes the i-th element of $\boldsymbol{v}$.

2.1 Additive Secret Sharing

In our 2PC protocols, we use 2-out-of-2 additive secret sharing over a ring R. A pair of values $[x] = ([x]_1, [x]_2) \in R^2$ is called an additive sharing of $x \in R$ if the parties P_1 and P_2 respectively hold $[x]_1$ and $[x]_2$ such that $x = [x]_1 + [x]_2$. Note that for any sharings $[x]$, $[y]$ and public value c, each share-holder can locally compute $[x + y]$, $[c + x]$, and $[cx]$. The scheme consists of the following protocols:

Share(x)**:** This is a local operation: Choose $r \xleftarrow{\$} R$ at random, and output $[x] = ([x]_1, [x]_2) := (r, x - r)$.

Reconst$([x])$**:** This is a two-party protocol executed by parties P_1 and P_2 who respectively hold $[x]_1$ and $[x]_2$: P_i sends $[x]_i$ to P_{i+1}, and both parties output $[x]_1 + [x]_2$.

For a vector $\boldsymbol{x} = (x_1, x_2, \ldots)$, we write $[\boldsymbol{x}]_i$ to denote a vector $([x_1]_i, [x_2]_i, \ldots)$ of shares. We may use additive secret sharings over different rings in the same protocol. In particular, when $R = \mathbb{Z}_2$, we explicitly write $[\cdot]^B$ and Share^B for boolean sharings.

2.2 Linear Group Action

Let R be a ring, V be an R-module, and $(G, \circ, 1)$ be a group. A (right) *linear group action* of G on V is a function $\psi : V \times G \to V$ satisfying

$$\begin{aligned} \psi(x, 1) &= x && \forall x \in V, \\ \psi(x, g_1 \circ g_2) &= \psi(\psi(x, g_1), g_2) && \forall x \in V, g_1, g_2 \in G, \\ \psi(x + y, g) &= \psi(x, g) + \psi(y, g) && \forall x, y \in V, g \in G, \\ r\psi(x, g) &= \psi(rx, g) && \forall r \in R, x \in V, g \in G. \end{aligned}$$

We will write the element $\psi(x, g) \in V$ simply as $x \cdot g$. A left linear group action is defined analogously.

Example 1. Place-permutation Action. Let G be the symmetric group $(S_n, \circ, \mathbf{1})$ of degree n. Here, $\pi_2 \circ \pi_1$ is given as the standard composition of functions, and 1 is the identity function. We define the function $\psi : R^n \times G \to R^n$ by

$$(\boldsymbol{v} \cdot \pi)_i := \boldsymbol{v}_{\pi(i)}.$$

As an example, if $\boldsymbol{v} = (7, 5, 3, 1)$ and $\pi = (1, 4, 2, 3)$ then $\boldsymbol{v} \cdot \pi = (7, 1, 5, 3)$. As easily checked, ψ is a (right) linear group action. We note that if $\{1, \ldots, n\} \subseteq R$, then there is the vector representation of $\pi \in S_n$ given by $(\pi(1), \pi(2), \ldots, \pi(n)) \in \{1, \ldots, n\}^n$, and thus we can regard $\pi \in R^n$. In this case, it holds that $\pi_1 \cdot \pi_2 = \pi_1 \circ \pi_2$. [4]

Example 2. Circular Shift Action. Let G be the cyclic group of degree n, *i.e.,* $G = (\mathbb{Z}_n, +, 0)$. We define the function $\psi : R^n \times G \to R^n$ as follows:

$$(\boldsymbol{v} \cdot g)_i := \boldsymbol{v}_{i+g}$$

where the addition $i + g$ is modulo n. For example, if $\boldsymbol{v} = (7, 5, 3, 1)$ and $g = 1$ then $\boldsymbol{v} \cdot g = (5, 3, 1, 7)$. This determines the circular shift action, which is a linear group action on R^n.

Example 3. XOR Permutation Action. Let G be the commutative ring $\mathbb{Z}_2^m$. We define $\psi : R^{2^m} \times G \to R^{2^m}$ as follows:

$$(\boldsymbol{v} \cdot g)_i := \boldsymbol{v}_{i \oplus g}$$

where $i \oplus g \in \{1, \ldots, 2^m\}$ is defined so that the binary representation of $(i \oplus g) - 1$ is given as the bitwise XOR of $i - 1$ and $(g_m, \ldots, g_1)$. For example, if $\boldsymbol{v} = (7, 5, 3, 1)$ and $g = (0, 1)$ then $\boldsymbol{v} \cdot g = (5, 7, 1, 3)$. We can easily check that ψ is a linear group action.

Example 4. General Linear Group Action. Let $G = GL_n(R)$, which consists of the set of $n \times n$ invertible matrices with entries from R and matrix multiplication. There are two linear group actions of G on different modules, $G \times R^n \to R^n$ and $GL_n(R) \times G \to GL_n(R)$, given by ordinary matrix multiplication:

$$R \cdot \boldsymbol{v} := R\boldsymbol{v}, \quad M \cdot R := MR.$$

In general, when V is a free R-module (*e.g.,* $V = R^n$), there is a one-to-one correspondence between linear group actions of G and linear

[4] In the introduction, it can be said that we have abused the notation by writing $\pi(\boldsymbol{v})$ to mean $\boldsymbol{v} \cdot \pi$. This is an abuse in the first place since an automorphism π is only formally defined on $[n] \to [n]$ (but not on $R^n \to R^n$, and we abuse to do so).

representations of G (*i.e.,* group homomorphisms $G \to GL(V)$). Note that the prior examples are specific cases.

More Examples. Due to limited space, we provide more examples of linear group action in Appendix C.1.

Encoding Group Elements. In some of our protocols, it will be convenient to encode group elements in a form that is compatible with additive sharing. Assume $|G| \leq |V|$. We call an injective function $e : G \to V$ an encoding of G to V if $e(g) \cdot r = e(g \circ r)$ for all $g, r \in G$. The encodings for the four examples are as follows.

- For the place-permutation action over n elements, the encoding of $g \in S_n$ is its vector representation.
- For the circular shift action over n elements, the encoding of $g \in \mathbb{Z}_n$ is the g-th unit vector, denoted as $\boldsymbol{e}^{(g)} := (0, \ldots, 0, 1, 0, \ldots, 0)$ with 1 at the g-th position.
- For the XOR permutation action, the encoding of $g \in \mathbb{Z}_2^m$ is the $\tilde{g}$-th unit vector where $\tilde{g} = \sum_{i=1}^{m} g_i 2^{i-1} + 1$.
- For the (right) general linear group action, the encoding is simply the identity function.

Note that for all these cases, the inverse e^{-1} is efficiently computable. When the context is clear, for simplicity of protocol descriptions, we often write $e(g)$ merely as g itself.

2.3 Security of Protocols

In this paper, we only treat security for 2PC protocols in the presence of a static semi-honest (passive) corruption of a single party. Since its definition is standard, we defer the formal definition to the full version.

We describe and prove the security of our protocols using the hybrid model, where parties run in a protocol with messages and also have access to a trusted party computing a subfunctionality for them. When the subfunctionality is $\mathcal{G}$, we say that the protocol works in the $\mathcal{G}$-hybrid model.

For notational convenience, for a protocol Π in the $\mathcal{G}$-hybrid model for some two-party functionality $\mathcal{G}$, when we write "$(y_1, y_2) \leftarrow \mathcal{G}(x_1, x_2)$" in the description of Π, we mean that each party P_i sends the input x_i to the subfunctionality $\mathcal{G}$, and receives y_i from $\mathcal{G}$. If $\mathcal{G}$ is a functionality that takes a sharing $[x] = ([x]_1, [x]_2)$ and outputs another sharing $[y] = ([y]_1, [y]_2)$, we may also write "$[y] \leftarrow \mathcal{G}([x])$". We also extend this notation for functionalities taking multiple shareings $[x_1], [x_2], \ldots$ as input and outputs multiple sharings $[y_1], [y_2], \ldots$, and write "$([y_1], [y_2], \ldots) \leftarrow \mathcal{G}([x_1], [x_2], \ldots)$".

In this paper, due to the lack of space, we show the correctness of our proposed protocols and state its security theorems but defer the formal security proofs to the full version.

2.4 Offline/Online Paradigm

As mentioned in the introduction, we consider offline (pre-processing) and online phase. As a recurring convention throughout the paper, a full protocol for functionality X will use a sub-functionality, say X-prep, which returns the required correlated randomness for X. A call to X-prep is realized in the pre-processing phase on one of the following two ways.

- *Pre-processing protocol.* We run a 2-party protocol realizing X-prep.
- *Client-aided setting.* We consider a semi-honest third party (a client) who generates the output as specified in X-prep, and sends the output to the two parties.

In the paper body, we mainly focus on the client-aided approach, which allows us to obtain very efficient protocols. We also propose protocols for the pre-processing phases for the protocols used in this paper. Due to limited space, we defer them to the full version.

3 OBLIVIOUS LINEAR GROUP ACTIONS

In this section, we present the functionality definition and our protocol for oblivious linear group actions (LGA) of G on V.

We first introduce a core structure: *double-sharings* of group elements in Section 3.1. In Section 3.2, we give the formal (extended) definition of the functionality for oblivious linear group actions LGA. In Section 3.3, we explain the correlated randomness used in our protocol. In Section 3.4, we present our proposed protocol for LGA. In Section 3.5, we explain concrete instances of our protocol for LGA. In Section 3.6, we present a variant of our protocol for LGA when one of the inputs is a public (plaintext) value.

3.1 Double-Sharing of Group Elements

Double-Sharings. Let $(G, \circ)$ be a group. A *double-sharing* of $g \in G$, denoted by $\langle g \rangle$, consists of $(\langle g \rangle_1, \langle g \rangle_2)$ where $\langle g \rangle_i := (g_i, g_i') \in G^2$ and the following holds:

$$g = g_1 \circ g_2' = g_2 \circ g_1'. \tag{1}$$

Each $\langle g \rangle_i$ is called a *double-share* of g. We denote the process of randomly generating a double-sharing of g by $\langle g \rangle \leftarrow \mathsf{DShare}_G(g)$. Namely, from g, we randomly choose $g_1, g_1' \xleftarrow{\$} G$ and compute g_2, g_2' according to Eq.(1). We may also slightly abuse the notation and write "$\langle g \rangle \in \mathsf{DShare}_G(g)$" to mean that $\langle g \rangle$ is a possible double-sharing of $g \in G$.

Single-Sharings. [5] In addition to double-sharings, it is useful to also consider "single"-sharings of group elements. For a group element $g \in G$, we define a *left-to-right single-sharing (LR-sharing)* $|g\rangle$ and a *right-to-left single-sharing (RL-sharing)* $\langle g|$ by

$$|g\rangle = (|g\rangle_1, |g\rangle_2) \quad \text{s.t.} \quad |g\rangle_1 \circ |g\rangle_2 = g,$$
$$\langle g| = (\langle g|_1, \langle g|_2) \quad \text{s.t.} \quad \langle g|_2 \circ \langle g|_1 = g.$$

Local Conversions. We provide some useful local conversions.

- From $\langle g \rangle$, the parties can locally compute $\langle g^{-1} \rangle$ by simply setting $\langle g^{-1} \rangle_i := (g_i'^{-1}, g_i^{-1})$.
- From $\langle g \rangle$ and a public $h \in G$, the parties can locally compute $\langle h \circ g \rangle$ and $\langle g \circ h \rangle$ by setting

$$\langle h \circ g \rangle_i := (h \circ g_i, g_i') \quad \text{and} \quad \langle g \circ h \rangle_i := (g_i, g_i' \circ h).$$

  Note that $\langle g \circ h \rangle$ is in general *not* locally computable from $\langle g \rangle, \langle h \rangle$.
- From $\langle g \rangle$, the parties can locally compute $|g\rangle$ by setting $|g\rangle_1 = g_1$ and $|g\rangle_2 = g_2'$. Similarly, one compute $\langle g|$ by setting $\langle g|_1 = g_1'$ and $\langle g|_2 := g_2$.

Simplification in Commutative Groups. In case of commutative groups, we simplify the definition for double-sharings by setting $g_1 = g_1'$ and $g_2 = g_2'$, and write the share of each party as $\langle g \rangle_1 = g_1$ and $\langle g \rangle_2 = g_2$. Hence, in this case, single-sharings and

[5] These will be used in our LGA protocol for public values presented in Section 3.6.

double-sharings are identical. Moreover, $\langle g \circ h \rangle$ can be locally computed from $\langle g \rangle, \langle h \rangle$. Generating a random share for a *random* $g \in G$ can be done *non-interactively* by each party: we let P_i choose $g_i \xleftarrow{\$} G$ and define $g = g_1 \circ g_2$.

3.2 Formal Definition of the Functionality

We give the formal definition of the functinality for oblivious linear group actions, denoted by LGA. To capture the "re-invocability" of group actions r given in the form of a double-sharing, LGA is divided into two cases: If the auxiliary input is $\perp$, LGA outputs $[x \cdot r]$ for a *new* random element $r \in G$. Otherwise, it outputs $[x \cdot r]$ for a *previously generated* $\langle r \rangle$, provided as the auxiliary input. The functionality also outputs $\langle r \rangle$ for future invocations. More formally, the functionality LGA is defined as follows:

Functionality LGA

Input: $(([x]_1, aux_1), ([x]_2, aux_2))$ where $x \in V$, and $(aux_1, aux_2) = (\perp, \perp)$ or $(aux_1, aux_2) = \langle r \rangle \in \mathsf{DShare}_G(r)$ for some $r \in G$.

Output: $(([x \cdot r]_1, \langle r \rangle_1), ([x \cdot r]_2, \langle r \rangle_2))$.

(1) If $(aux_1, aux_2) = (\perp, \perp)$, then sample $r \xleftarrow{\$} G$ and let $\langle r \rangle \leftarrow \mathsf{DShare}_G(r)$. Else, reconstruct $r \in G$ from $(aux_1, aux_2) = \langle r \rangle$.
(2) Compute $x := [x]_1 + [x]_2$.
(3) Compute $[x \cdot r] \leftarrow \mathsf{Share}(x \cdot r)$.
(4) Return $(([x \cdot r]_1, \langle r \rangle_1), ([x \cdot r]_2, \langle r \rangle_2))$.

Notation when used as a subfunctionality:
If $(aux_1, aux_2) = (\perp, \perp)$: $([x \cdot r], \langle r \rangle) \leftarrow \mathsf{LGA}([x])$;
If $(aux_1, aux_2) = \langle r \rangle$: $[x \cdot r] \leftarrow \mathsf{LGA}([x]; \langle r \rangle)$.
(We omit $\langle r \rangle$ from the output to avoid redundancy.)

3.3 Correlated Randomness for LGA

We now define the correlated randomness prepared for using in our protocol for LGA. The basic intuition follows from our explanation in the case of shuffles in Section 1.1; here we generalize to the case of linear group actions. Consider a linear group action of G on V. To compute a sharing of the action from two sharings $[x]$ and $\langle r \rangle$, we prepare the following correlated randomness: Each party P_i holds $(\langle r \rangle_i, a_i, a'_i) \in G^2 \times V^2$ where $a_i, a'_i \in V$ satisfy the following: $a'_1 = a_2 \cdot r'_1 + c$, $a'_2 = a_1 \cdot r'_2 - c$ for some $c \in V$.

We can verify that each party cannot obtain any information about the other party's share without knowing the secret r. Note that while a_i, a'_i are used as one-time pads, the double-sharing $\langle r \rangle$ is reusable as long as the value of r does not leak.

The functionality that generates correlated randomness is defined as LGA-prep described below. Each party's input is either a double-share $\langle r \rangle_i$ or $\perp$ (the symbol representing "no-input"). Similarly to LGA, LGA-prep is divided into two cases:

- If the input is empty $\perp$, LGA-prep generates a correlated randomness containing a double-sharing $\langle r \rangle$ of a new random $r \in G$.
- Else, LGA-prep uses the input $\langle r \rangle_i$ and generates only a_i, a'_i.

Functionality LGA-prep

Input: (aux_1, aux_2) such that $(aux_1, aux_2) = (\perp, \perp)$ or $(aux_1, aux_2) = \langle r \rangle \in \mathsf{DShare}_G(r)$ for some $r \in G$

Output: $((\langle r \rangle_1, a_1, a'_1), (\langle r \rangle_2, a_2, a'_2))$

(1) If $(aux_1, aux_2) = (\perp, \perp)$, then sample $r \xleftarrow{\$} G$ and compute $\langle r \rangle \leftarrow \mathsf{DShare}_G(r)$, where $\langle r \rangle_i = (r_i, r'_i)$. Otherwise, regard (aux_1, aux_2) as a double-sharing $\langle r \rangle$.
(2) Sample $a_1, a_2, c \xleftarrow{\$} V$.
(3) Compute $a'_1 := a_2 \cdot r'_1 + c$ and $a'_2 := a_1 \cdot r'_2 - c$.
(4) Return $((\langle r \rangle_1, a_1, a'_1), (\langle r \rangle_2, a_2, a'_2))$.

Generating the above correlated randomness is trivial in the client-aided setting: a client simply generates and distributes all the randomness for the two computing parties. We can also realize the above functionality in the standard two-party setting by making use of the technique of [9]; we provide this in the full version.

3.4 Our Protocol for Oblivious LGA

We now present our protocol Π_{LGA} for computing LGA, which uses LGA-prep as a subfunctionality:

Protocol 1. Π_{LGA}

Input: $([x]_i, aux_i)$ from P_i, where $x \in V$ and $(aux_1, aux_2) = (\perp, \perp)$ or $(aux_1, aux_2) = \langle r \rangle \in \mathsf{DShare}_G(r)$ for some $r \in G$.

Output: $([y]_i, \langle r \rangle_i)$ for P_i.

Subfunctionality: LGA-prep.

(1) The parties call LGA-prep(aux_1, aux_2), and each party P_i receives $(\langle r \rangle_i, a_i, a'_i)$, where $\langle r \rangle_i = (r_i, r'_i)$.
(2) Each party P_i locally computes $v_i := [x]_i \cdot r_i + a_i$, and sends it to P_{i+1}.
(3) Each party P_i locally computes $[y]_i := v_{i+1} \cdot r'_i - a'_i$, and outputs $([y]_i, \langle r \rangle_i)$.

The correctness of the protocol is shown as follows:

$$[y]_1 = v_2 \cdot r'_1 - a'_1 = ([x]_2 \cdot r_2 + a_2) \cdot r'_1 - (a_2 \cdot r'_1 + c) = [x]_2 \cdot r - c,$$
$$[y]_2 = v_1 \cdot r'_2 - a'_2 = ([x]_1 \cdot r_1 + a_1) \cdot r'_2 - (a_1 \cdot r'_2 - c) = [x]_1 \cdot r + c,$$

where the last equalities use $r_2 \circ r'_1 = r_1 \circ r'_2 = r$. Note that we have $[y]_1 + [y]_2 = ([x]_1 + [x]_2) \cdot r = x \cdot r$. We note that the protocol for left linear group actions can be defined similarly.

We can divide the protocol into the offline/online phases. If $\langle r \rangle$ is newly generated or can be determined before the online phase, then Step 1 of Π_{LGA} can be executed during the offline phase. In the online phase, the protocol takes only one round, and each party sends an element of V to the other. The security of Π_{LGA} is guaranteed as follows.

Theorem 3.1. *The protocol* Π_{LGA} *for* LGA *is perfectly semi-honest secure in the* LGA-prep*-hybrid model.*

3.5 Instances of Our Main Protocol

Oblivious Shuffle. A typical instance of LGA is oblivious shuffling. We write Shuffle and Π_{Shuffle} for the functionality and the corresponding protocol, respectively, in the case the linear group action is the place-permutation action (see Example 1 in Section 2.2).

When $\pi \in G$ is chosen uniformly at random, Shuffle coincides with the ordinary oblivious shuffling functionality. On the other hand, we can also use π depending on previous executions of

Π_{Shuffle}. In particular, by setting the auxiliary input (aux_1, aux_2) to be a double-sharing $\langle \pi^{-1} \rangle$, our shuffling protocol enables us to use the unshuffling technique, which is known to be effective in reducing communication complexity.

Oblivious Circular Shift (Roulette) and XOR-Shuffle. When the place-permutation action of a symmetric group is excessive, we can use a "light" variant of Shuffle. We write Roulette and Π_{Roulette} for the functionality and the corresponding protocol, respectively, in the case the linear group action is the circular shift action (see Example 2 in Section 2.2). Similarly, in the case of the XOR permutation action (see Example 3 in Section 2.2), we write XORShuffle and $\Pi_{\mathsf{XORShuffle}}$. In these cases, since G is commutative, we can save the memory cost for a correlated randomness because a double-sharing of an element of G consists only of a single element of G. Also, since r_1 and r_2 in $\langle r \rangle$ form an additive secret-sharing, we can reduce the round complexity for some protocols compared to using Π_{Shuffle}.

Oblivious Matrix Multiplications. When considering general linear group actions, the functionality LGA provides secure matrix multiplication by a random invertible matrix. Although the memory cost for correlated randomness is higher than the above two cases, we can consider any automorphism of R^n in a uniform way. In other words, it is more general than the above two. Moreover, by combining LGA for left and right linear group actions, we obtain a round-efficient protocol in several cases; we elaborate on this in Sections 4.4 and 5.3.

More Instances and Some Optimizations. Due to limited space, we provide more instances of our main protocol in Appendix C.2. Moreover, in many cases, we can use the "row-reduction technique" and reduce the number of elements each party sends in Π_{LGA} by one element. See the full version for details.

3.6 Oblivious LGA for Public Values

Consider the cases where only one party holds a secret value $x \in V$ as input, or the parties have a public value x, and wish to compute $[x \cdot r]$. In these cases, we can reduce the communication and memory cost of the protocol Π_{LGA}. For the case where P_1 holds the input value x, the corresponding functionality LGA_p is as follows.

Functionality LGA_p

Input: $((x, aux_1), aux_2)$, where $x \in V$ and $(aux_1, aux_2) = (\bot, \bot)$ or $(aux_1, aux_2) = |r\rangle$ for some $r \in G$.

Output: $(([x \cdot r]_1, |r\rangle_1), ([x \cdot r]_2, |r\rangle_2))$.

(1) If $(aux_1, aux_2) = (\bot, \bot)$, then sample $r \xleftarrow{\$} G$ and generate a random LR-sharing $|r\rangle$. Else, reconstruct r from $|r\rangle$.
(2) Compute $[x \cdot r] \leftarrow \mathsf{Share}(x \cdot r)$.
(3) Return $(([x \cdot r]_1, |r\rangle_1), ([x \cdot r]_2, |r\rangle_2))$.

Notation when used as a subfunctionality:

If $(aux_1, aux_2) = (\bot, \bot)$: $([x \cdot r], |r\rangle) \leftarrow \mathsf{LGA\_p}(x)$;
If $(aux_1, aux_2) = |r\rangle$: $[x \cdot r] \leftarrow \mathsf{LGA\_p}(x; |r\rangle)$.
(We omit $|r\rangle$ from the output to avoid redundancy.)

Similarly to LGA, LGA_p allows to re-invoke an existing sharing of r. However, unlike LGA, it operates with an LR-sharing rather than a double-sharing. We defer its protocol to the full version.

4 VARIANTS OF SECURE GROUP ACTIONS

Section 3 allows us to securely apply group actions specified by $g \in G$ whether g is a newly randomized element or a previously generated one. To apply a previous element, we let the parties input the *double-sharing* format of g, through an auxiliary input. In this section, we provide another useful interface to apply a group action with a previous element in the *additive sharing* format of g. This is motivated by applications whose sub-protocols output additive sharing of g, which will then be used as an input in applying group actions in another sub-protocol, and this is where our interface comes into play. Interestingly, the protocols with the new interface will in turn be constructed based on our previous LGA (and LGA_p) functionalities, via the so-called "act-then-reveal" technique.

4.1 Act-then-Reveal Approach

In general, to securely compute $[f(x)]$ from $[x]$ in a secret-sharing-based MPC framework, we often use the following "randomize-then-reveal" technique:

(1) Parties randomize the input $[x]$ with some kind of correlated randomness for the function f in some way and receive a sharing $[y]$ of the randomized secret y.
(2) Parties reveal the randomized secret y. This leaks no information about x because of randomization.
(3) Parties compute $[f(x)]$ in some way from the randomized secret y and the correlated randomness.

This technique is useful in the offline/online paradigm, because correlated randomness is data-independent and can be generated in the offline phase. A typical example of the technique is the Beaver triple technique for secure multiplication.

Consider the case where f is a function related to a linear group action. To realize Steps 1 and 2 for a group element, we generalize the shuffle-then-reveal approach, implicitly used in [11], to what we call "act-then-reveal" approach.[6] The main idea of this approach is to use a numerical representation of $g \in G$, namely $e(g)$ (see Section 2.2) and to use LGA as a randomizer by letting an additive sharing of $e(g)$ itself be the *main input* of LGA (rather than the auxiliary input). More formally, the process is described as follows:

(1) On input $[e(g)] \in V$, the parties invoke $([v], \langle r \rangle) \leftarrow \mathsf{LGA}([e(g)])$.
(2) The parties compute $v := \mathsf{Reconst}([v])$, and output $(v, \langle r \rangle)$.

Then it holds that $v = e(g \circ r)$ for a uniformly random $r \in G$, so the parties can obtain the randomized group element $g' := g \circ r \in G$ as e is injective. By the surjectivity of the multiplication by r, the value of v leaks no information on g.

4.2 Secure Inverse Group Actions

We start with the most useful way to re-invoke group actions, namely, applying the *inverses*. For $x \in V$, $g \in G$, the functionality is defined as: $\mathsf{Share}(x \cdot g^{-1}) \leftarrow \mathsf{LGA\_inverse}([x], [e(g)])$. Here, we observe that g is input in the additive sharing format, namely, $[e(g)]$. We present the protocol $\Pi_{\mathsf{LGA\_inverse}}$ for realizing LGA_inverse, which uses LGA as a subfunctionality:

[6] The previous approach of [11] consider only shuffles, while ours can deal with any linear group actions.

Protocol 2. $\Pi_{\mathsf{LGA\_inverse}}$

Input: $([x]_i, [e(g)]_i)$ from P_i, where $x \in V$ and $g \in G$.
Output: $[y]_i$ for P_i, where $[y] \leftarrow \mathsf{Share}(x \cdot g^{-1})$.
Subfunctionality: LGA.

(1) The parties invoke $(([e(h)], \langle r \rangle) \leftarrow \mathsf{LGA}([e(g)])$.
(2) The parties invoke $[z] \leftarrow \mathsf{LGA}([x]; \langle r \rangle)$.
(3) The parties compute $h := e^{-1}(\mathsf{Reconst}([e(h)]))$.
(4) Each P_i locally computes and outputs $[y]_i := [z]_i \cdot h^{-1}$.

Since the online phase of Steps 1-2 can be run in parallel, this protocol takes two online rounds. Note that while the correlated randomness $\langle r \rangle$ appears in both steps, it is generated in the preprocessing. The correctness is shown as follows:

$$y = z \cdot h^{-1} = (x \cdot r) \cdot (g \circ r)^{-1} = x \cdot (r \circ r^{-1} \circ g^{-1}) = x \cdot g^{-1}.$$

This protocol is a generalized version of the protocol of Bogdanov *et al.* [4], which is defined for the place-permutation action. We note that this construction works for any oblivious shuffling protocol.

The security of $\Pi_{\mathsf{LGA\_inverse}}$ is guaranteed by the following theorem.

Theorem 4.1. *The protocol* $\Pi_{\mathsf{LGA\_inverse}}$ *for* LGA_inverse *is perfectly semi-honest secure in the* LGA*-hybrid model.*

4.3 Secure Group Actions in Additive Sharing

We now provide a protocol for securely applying a group action by an arbitrary group element that is input with the additive sharing format. We use what we call "un-acting" technique, generalized from "un-shuffling" method, used in [11].

For a preparation, we first describe the protocol for converting an additive sharing $[e(g)]$ to a random-double-sharing. Let A2D be the functionality that takes $[e(g)]$ (where $g \in G$) as input, and returns a random double-sharing $\langle g \rangle = (\langle g \rangle_1, \langle g \rangle_2) \leftarrow \mathsf{DShare}_S(g)$. Then, the protocol Π_{A2D} for computing A2D, which uses LGA as a subfunctionality, is as follows. It takes two online rounds.

Protocol 3. Π_{A2D}

Input: $[e(g)]_i$ from P_i, where $g \in G$.
Output: $\langle g \rangle_i$ for P_i, where $\langle g \rangle \leftarrow \mathsf{DShare}_G(g)$.
Subfunctionality: LGA.

(1) The parties invoke $([e(h)], \langle r \rangle) \leftarrow \mathsf{LGA}([e(g)])$.
(2) The parties compute $h := e^{-1}(\mathsf{Reconst}([e(h)]))$.
(3) Each party P_i locally computes $\langle g \rangle_i := (h \circ r_i'^{-1}, r_i^{-1})$ and outputs it.

The correctness holds as

$$g_1 \circ g_2' = (h \circ r_1'^{-1}) \circ r_2^{-1} = (g \circ r) \circ r^{-1} = g,$$
$$g_2 \circ g_1' = (h \circ r_2'^{-1}) \circ r_1^{-1} = (g \circ r) \circ r^{-1} = g,$$

where, intuitively, we operate the "un-acting" via r^{-1}.

The security of Π_{A2D} is guaranteed by the following theorem.

Theorem 4.2. *The protocol* Π_{A2D} *for* A2D *is perfectly semi-honest secure in the* LGA*-hybrid model.*

Note that correlated randomness containing the output $\langle g \rangle$ of Π_{A2D} can be generated in the offline phase, although $\langle g \rangle$ depends on the online input $[e(g)]$. In this case, the second entry of $\langle g \rangle_i$ of each party is the same as that of a pre-generated double-share $\langle r^{-1} \rangle_i$. Accordingly, by the definition of correlated randomness for LGA, if $(\langle r^{-1} \rangle_i, a_i, a_i')$ is correlated randomness, then so is $(\langle g \rangle_i, a_i, a_i')$. Hence we can generate such (a_i, a_i') in offline and use it as a part of correlated randomness $(\langle g \rangle_i, a_i, a_i')$.

We now present a protocol for applying a group element to a data where both are in the additively shared form. For $x \in V$ and $g \in G$, the functionality is defined as: $\mathsf{Share}(x \cdot g) \leftarrow \mathsf{LGA\_additive}([x], [e(g)])$. The protocol $\Pi_{\mathsf{LGA\_additive}}$ for computing LGA_additive, which uses A2D and LGA as subfunctionalities, is as follows:

Protocol 4. $\Pi_{\mathsf{LGA\_additive}}$

Input: $([x]_i, [e(g)]_i)$ from P_i, where $x \in V$ and $g \in G$.
Output: $[x \cdot g]_i$ for P_i, where $[x \cdot g] \leftarrow \mathsf{Share}(x \cdot g)$.
Subfunctionality: A2D, LGA.

(1) The parties invoke $\langle g \rangle \leftarrow \mathsf{A2D}([e(g)])$.
(2) The parties invoke $[x \cdot g] \leftarrow \mathsf{LGA}([x]; \langle g \rangle)$.
(3) Each party P_i outputs $[x \cdot g]_i$.

As mentioned above, the correlated randomness required in Step 2 is computable in the offline phase. Thus, this protocol takes three rounds in the online phase. The security of $\Pi_{\mathsf{LGA\_additive}}$ is guaranteed as follows.

Theorem 4.3. *The protocol* $\Pi_{\mathsf{LGA\_additive}}$ *for* LGA_additive *is perfectly semi-honest secure in the* (A2D, LGA)*-hybrid model.*

4.4 Secure Bidirectional Group Actions

We observe that a more round-efficient protocol for applying an additively shared group element to shared values is possible in the case both left and right group actions are defined. We call such group actions as bidirectional ones. Notable examples are actions by the general linear groups, *i.e.,* matrix multiplications.

In this subsection, we provide a functionality that securely applies the *left* group actions by g from the additive sharing of g with respect to the *right* group action. Let e be a numerical representation of G with respect to the *right* group action. We denote the functionalities for the left and right group actions by LGA_L and LGA_R, respectively.[7] The functionality LGA_bidirect is defined as: $\mathsf{Share}(g \cdot x) \leftarrow \mathsf{LGA\_bidirect}([x], [e(g)])$. The protocol $\Pi_{\mathsf{LGA\_bidirect}}$ for computing LGA_bidirect, which uses LGA_R and LGA_L as subfunctionalities, is as follows:

Protocol 5. $\Pi_{\mathsf{LGA\_bidirect}}$

Input: $([x]_i, [e(g)]_i)$ from P_i, where $x \in V$ and $g \in G$.
Output: $[g \cdot x]_i$ for P_i, where $[g \cdot x] \leftarrow \mathsf{Share}(g \cdot x)$.
Subfunctionality: LGA_R, LGA_L.

(1) The parties invoke $([e(h)], \langle r \rangle) \leftarrow \mathsf{LGA\_R}([e(g)])$.
(2) Each party P_i locally computes $\langle r^{-1} \rangle_i$ from $\langle r \rangle_i$.
(3) The parties compute $h := e^{-1}(\mathsf{Reconst}([e(h)]))$.
(4) The parties invoke $[z] \leftarrow \mathsf{LGA\_L}([x]; \langle r^{-1} \rangle)$.
(5) Each P_i locally computes $[y]_i := h \cdot [z]_i$ and outputs it.

Steps 3 and 4 can be run in parallel because the inputs are independent. Thus, this protocol takes two rounds in the online phase. The correctness is shown as follows:

$$y = h \cdot z = (g \circ r) \cdot (r^{-1} \cdot x) = g \cdot x.$$

[7] Note that LGA_R is LGA itself. A protocol for computing LGA_L can be constructed analogously to the protocol Π_{LGA}.

The security of $\Pi_{\mathsf{LGA\_bidirect}}$ is guaranteed as follows. We omit its proof since it is similar to the proof of Theorem 4.1.

Theorem 4.4. *The protocol* $\Pi_{\mathsf{LGA\_bidirect}}$ *for* LGA_bidirect *is perfectly semi-honest secure in the* (LGA_L, LGA_R)*-hybrid model.*

5 SIMPLE APPLICATIONS

This section presents three simple but useful applications of our oblivious linear group actions and their variants. The first two subsections on oblivious selection and oblivious unit-vectorization involve computation related to "positions" in vectors, and use our oblivious circular shifts (Roulette, Roulette_p) as building blocks. A recurring idea is again the act-then-reveal approach: randomly circularly shift the vector by the amount of such a position, reveal, compute, and shift back. Thanks to the feature that a double sharing for circular shifting is equivalent to an additive sharing, some communication required for oblivious linear group action can be omitted, and we obtain online-round-optimal protocols for these. The third section presents secure "batch" matrix multiplications, which are direct applications of secure bidirectional LGA.

We note that by considering other linear group actions, we can obtain several other applications by using the construction method proposed in this section. More applications obtained by this method, including oblivious multiplexer, oblivious polynomial evaluation, secure multiplication, can be found in Appendix D.

In the first two subsections, we consider the circular shift action of the group $\mathbb{Z}_N = \{1, \ldots, N\}$, and use an additive secret sharing over $\mathbb{Z}_N$. We write $[\cdot]^N$ for a sharing in the secret sharing scheme.

5.1 Oblivious Selection

We present a protocol for oblivious selection of an element from a secret-shared list $[\boldsymbol{v}]$ where $\boldsymbol{v} \in D^N$ for some domain D and $N \in \mathbb{N}$. The functionality is defined as: $\mathsf{Share}(\boldsymbol{v}_n) \leftarrow \mathsf{Select}([\boldsymbol{v}], [n]^N)$, where $n \in \mathbb{Z}_N$. In other words, it is the "shared" variant of 1-out-of-N oblivious transfer. The protocol Π_{Select} for computing Select, which uses Roulette as a subfunctionality, is as follows:

Protocol 6. Π_{Select}

Input: $([\boldsymbol{v}]_i, [n]_i^N)$ from P_i, where $\boldsymbol{v} \in D^N$ and $n \in \mathbb{Z}_N$.
Output: $[\boldsymbol{v}_n]_i$ for P_i, where $[\boldsymbol{v}_n] \leftarrow \mathsf{Share}(\boldsymbol{v}_n)$.
Subfunctionality: Roulette.

(1) Each party P_i samples $r_i \xleftarrow{\$} \mathbb{Z}_N$ and sets $\langle r \rangle_i := r_i$.
(2) The parties invoke $[\boldsymbol{y}] \leftarrow \mathsf{Roulette}([\boldsymbol{v}]; \langle r \rangle)$.
(3) Each party P_i locally computes $[m]_i^N := [n]_i^N - \langle r \rangle_i$.
(4) The parties compute $m := \mathsf{Reconst}([m]^N)$.
(5) Each P_i locally computes $[z]_i := [\boldsymbol{y}_m]_i$ and outputs it.

The correctness holds since: $z = \boldsymbol{y}_m = (\boldsymbol{v} \cdot r)_{n-r} = \boldsymbol{v}_n$. Recall that in the case of Roulette, the group action is commutative and a double-sharing $\langle r \rangle$ degenerates to just an additive sharing of $r \in \mathbb{Z}_N$. This allows each party P_i to choose $\langle r \rangle_i$ independently, as in Step 1. In addition, although this protocol is a special case of $\Pi_{\mathsf{LGA\_inverse}}$, this protocol takes only one online-round by running Steps 2 and 3-4 in parallel.

The security of Π_{Select} is guaranteed as follows. We omit its proof since it is similar to the proof of Theorem 4.1.

Theorem 5.1. *The protocol* Π_{Select} *for computing* Select *is perfectly semi-honest secure in the* Roulette*-hybrid model.*

We note that by using XORShuffle instead of Roulette in Π_{Select}, a similar construction gives the one-online-round protocol Π_{Mux} for computing secure multiplexer. See Section D.1 for details.

5.2 Oblivious Unit-Vectorization

We next consider secure conversion from integer to its "one-hot" encoding, which appeals to machine-learning literature [10, 42]. The functionality is defined as: $\mathsf{Share}(\boldsymbol{e}^{(n)}) \leftarrow \mathsf{Unitv}([n]^N)$, where $n \in \mathbb{Z}_N$ and recall that $\boldsymbol{e}^{(n)}$ denotes the n-th unit vector (in N-dimension). We present the protocol Π_{Unitv} for computing Unitv, which uses Roulette_p as a subfunctionality[8], as follows. We note that the notation '$\cdot$' here is a circular shift action (and not to be confused with integer multiplications).

Protocol 7. Π_{Unitv}

Input: $[n]_i^N$ from P_i, where $n \in \mathbb{Z}_N$.
Output: $[\boldsymbol{e}^{(n)}]_i^B$ for P_i, where $\boldsymbol{e}^{(n)}$ is the n-th unit vector and $[\boldsymbol{e}^{(n)}]^B \leftarrow \mathsf{Share}^B(\boldsymbol{e}^{(n)})$.
Subfunctionality: Roulette_p.

(1) The parties invoke $([\boldsymbol{b}]^B, |r\rangle) \leftarrow \mathsf{Roulette\_p}(\boldsymbol{e}^{(1)})$.
(2) Each party P_i locally computes $[m]_i^N := [n]_i^N - |r\rangle_i$.
(3) The parties compute $m := \mathsf{Reconst}([m]^N)$.
(4) Each party P_i locally computes and outputs $[z]_i^B := [\boldsymbol{b}]_i^B \cdot (m - 1)$.

Note that Step 1 can be offline, hence this protocol has one round in the online phase. The correctness is shown as follows:

$$\boldsymbol{z} = (\boldsymbol{e}^{(1)} \cdot r) \cdot (n - r - 1) = \boldsymbol{e}^{(1)} \cdot (n - 1) = \boldsymbol{e}^{(n)}.$$

The security of Π_{Unitv} is guaranteed as follows. We omit its proof since it is similar to the proof of Theorem 4.1.

Theorem 5.2. *The protocol* Π_{Unitv} *for computing* Unitv *is perfectly semi-honest secure in the* Roulette_p*-hybrid model.*

Applications of Oblivious Unit-Vectorization. Based on our online-one-round protocol for Unitv, we obtain some *online-one-round* protocols for some fundamental computations.

- *Interval test.* For any public $i, j \in \mathbb{Z}_N$ with $i \le j$, from $[x]^N$, we can compute $[(x \stackrel{?}{\in} \{i, \ldots, j\})]^B$ in one online-round. After running $\mathsf{Unitv}([x])$ once to get $[\boldsymbol{e}^{(x)}]^B$, the sum from i-th to j-th entries of $[\boldsymbol{e}^{(x)}]^B$ is locally computable, and results in the desired sharing since $(x \stackrel{?}{\in} \{i, \ldots, j\}) \Leftrightarrow \oplus_{i \le k \le j} (x \stackrel{?}{=} k)$ and $(x \stackrel{?}{=} k)$ corresponds to the value of $\boldsymbol{e}_k^{(x)}$.
- *Arithmetic-to-Boolean (A2B) share conversion.* An A2B conversion protocol takes $[x]$ as input, and outputs $([x_t]^B, \ldots, [x_1]^B)$, where $(x_t, \ldots, x_1)$ is the bit decomposition of x, *i.e.,* $x = \sum_{i=1}^{t} 2^{i-1} x_i$. We construct this by running $\mathsf{Unitv}([x])$ once, and (locally) performing interval tests properly. For example, the LSB, $[x_1]^B$, is obtained by the sum of equality tests to the odd entries of $\mathsf{Unitv}([x])$. Since we only use one execution of Unitv, we have optimal-online-round (1-round) protocol for A2B.

[8] As in Roulette, an LR-sharing $|r\rangle$ in the case of Roulette_p degenerates to an additive sharing of $r \in \mathbb{Z}_N$.

Our interval test and A2B protocols have very efficient online protocols, namely, each party sends only one $\mathbb{Z}_N$ element in one round. The downside is the large size of correlated randomness, linear in the size of one-hot encodings which is exactly N, and hence the domain size N is required to be small (*i.e.,* polynomial-size in the security parameter) for these. We note that optimal-online-round protocols in the pre-processing model for interval tests and A2B share conversion have also been achieved via the use of function secret sharing (FSS) [5–7]. The correlated randomness for these protocols only grows polylogarithmically in N, but also requires a polylogarithmic number of invocations of a PRG in the on-line phase. One difference is that our interval test and A2B protocols are information-theoretically secure, while the FSS-based schemes require a PRG and hence are computationally secure.

5.3 Secure Batch Matrix Multiplications

Batch matrix multiplications consider the situation of multiplying some secret-shared vectors $[\boldsymbol{v}^{(1)}], \ldots, [\boldsymbol{v}^{(k)}]$ securely by the same secret-shared $n \times n$ invertible matrix $[M]$. To do this, we directly use secure bidirectional group actions, $\Pi_{\mathsf{LGA\_bidirect}}$, in Section 4.4. In $\Pi_{\mathsf{LGA\_bidirect}}$, first we set g to be M and execute Steps 1-2 during one communication round. Then execute Step 3. At the same time, Steps 4-5 are executed for each i in parallel, setting x to $\boldsymbol{v}^{(i)}$. In this method, each party sends $2n^2$ elements in Steps 1 and 3, and n elements in Step 4 for each $[\boldsymbol{v}^{(i)}]$, so that the total number of elements sent is $2n^2 + kn$. When the vectors $\{[\boldsymbol{v}^{(i)}]\}$ are already given, the number of online rounds is two. Note that the correlated randomness for $r, r^{-1} \in GL_n(R)$ is ready from the pre-processing.

Let us compare with the naive method of computing $[M\boldsymbol{v}^{(i)}]$ by n^2 secure multiplications for each i. This method is round-optimal but requires kn^2 Beaver triples and $2kn^2$ elements sent. The online communication cost of our method corresponds to that in the case of using the specific form of Beaver triples obtained by the reusing technique, where the online communication cost is $2n^2 + kn$.

6 APPLICATION: RADIX SORT

In this section, we propose an oblivious sorting protocol in the two-party setting as an application of the protocol Π_{Shuffle}. We focus on the radix sort protocol since it has the advantage (compared to comparison-based sorting) that the number of communication rounds depends only on the bit length ℓ of each key. Our protocol is based on a similar approach to the three-party protocol by Chida *et al.* [11], but using our oblivious shuffle as a building block as well as introducing various optimizations shown in this section, we obtain a two-party protocol with a reduced round complexity.

6.1 Overview of Our Radix Sort Protocol

For preparation, we give some notations. Let N be the number of records, and let ℓ denote the bit length of each key. We assume that the domain R of arithmetic shares is $\mathbb{Z}_{2^\lambda}$ satisfying $2^\lambda \geq N$. The assumption guarantees the existence of the vector representation $\boldsymbol{e}$ of S_N.[9] We call a permutation $\sigma \in S_N$ the *stable sorting* of $\boldsymbol{x} \in \mathbb{Z}_{2^\ell}^N$, if $\boldsymbol{x} \cdot \sigma^{-1}$ is the sorted vector of $\boldsymbol{x}$ in a stable manner, that is, σ satisfies for all $i < j$,

$$(\boldsymbol{k} \cdot \sigma^{-1})_i \leq (\boldsymbol{k} \cdot \sigma^{-1})_j \ \wedge (\boldsymbol{k} \cdot \sigma^{-1})_i = (\boldsymbol{k} \cdot \sigma^{-1})_j \Longrightarrow \sigma_i^{-1} \leq \sigma_j^{-1}.$$

It means: if $\sigma_i = k$, then $\boldsymbol{x}_i$ is the k-th smallest element, and 'stable' means that the relative order of records with equal keys is preserved, so that such a permutation is uniquely determined from $\boldsymbol{x}$.

The functionality BooleanStableSort for stable sorting is defined as: $\mathsf{Share}(\sigma) \leftarrow \mathsf{BooleanStableSort}([\boldsymbol{k}^\ell]^B, \ldots, [\boldsymbol{k}^1]^B)$, where $\boldsymbol{k}^j \in \mathbb{Z}_2^N$ and σ is the stable sorting of $\sum_{j=1}^{\ell} 2^{j-1}\boldsymbol{k}^j$. Note that if we wish to sort keys in the form of arithmetic sharings, we need to execute a protocol for bit-decomposition before an invocation of BooleanStableSort. If we wish to make the input keys sorted, we need to execute $\Pi_{\mathsf{LGA\_inverse}}$ after BooleanStableSort.

To realize BooleanStableSort, we will construct a protocol for the following L-bit-wise radix sort algorithm.

(0) Divide the binary representation of the input $\boldsymbol{x} \in \mathbb{Z}_{2^\ell}^N$ into L bits from the lower digit and obtain $\boldsymbol{x}^K, \ldots, \boldsymbol{x}^1 \in \mathbb{Z}_{2^L}^N$.
(1) $\sigma_1 := GBP(\boldsymbol{x}^1)$, $\rho_1 := \sigma_1$.
(2) For each $i = 2, \ldots, K$ do

$$\boldsymbol{y}^i := \boldsymbol{x}^i \cdot \rho_{i-1}^{-1}, \qquad \sigma_i := GBP(\boldsymbol{y}^i), \qquad \rho_i := \sigma_i \circ \cdots \circ \sigma_1.$$

where $K = \lceil \frac{\ell}{L} \rceil$ and $GBP : \mathbb{Z}_{2^L}^N \to S_N$ is the function that returns the stable sorting $GBP(\boldsymbol{v}) \in S_N$ of the input $\boldsymbol{v} \in \mathbb{Z}_{2^L}^N$. An advantage of the algorithm is that we sort only the part that is to be the input of GBP in the next step, which saves the number of vectors to be permuted, as shown in [11]. This gives the stable sorting ρ_i of $\sum_{j=1}^{i} 2^{L(j-1)}\boldsymbol{x}^j$ for each i, and thus ρ_K is the stable sorting of $\boldsymbol{x}$.

Consider computing Steps 1-2 of the above algorithm in MPC. This requires protocols for computing Place-permutation action, Composition of permutations, and GBP. For the former two, we can use $\Pi_{\mathsf{LGA\_inverse}}$ and $\Pi_{\mathsf{LGA\_additive}}$, respectively. For GBP, we define the functionality GenBitsPerm for sorting L-bit integers, which takes $[\boldsymbol{k}^L]^B, \ldots, [\boldsymbol{k}^1]^B$ and $\gamma \in S_N$ as input, and returns a sharing $[\sigma]$ of the stable sorting of $\sum_{j=1}^{L} 2^{j-1}\boldsymbol{k}^j \cdot \gamma^{-1}$. To realize GenBitsPerm, we consider computing the *counting sort* algorithm in MPC, as in the protocols of [4] and [11].

6.2 Optimization 1: Precomputation for Counting Sorting

In order to reduce communication complexity, we use a precomputation method to generate special correlated randomness for secure counting sorting.

The functionality GBP-prep for the precomputation is described below. The correlated randomness we use is in the form of a combination of the correlated randomness for XORShuffle and a boolean variant of the unit vectorization protocol.

Functionality GBP-prep

(1) Locally compute $(\langle c \rangle_i, \boldsymbol{a}_i, \boldsymbol{a}'_i)_{i=1,2} \leftarrow$ XORShuffle-prep.
(2) Set $([c^L]_i^B, \ldots, [c^1]_i^B) := \langle c \rangle_i$.
(3) Compute $\boldsymbol{v} := \boldsymbol{e}^{(n)}$ for $n = \sum_{j=1}^{L} 2^{j-1}c^j + 1$.
(4) Compute the following by using Share:

$$\mathsf{CR}_i^L := (\langle c \rangle_i, [\boldsymbol{v}]_i, \boldsymbol{a}_i, \boldsymbol{a}'_i)$$

(5) Return $(\mathsf{CR}_1^L, \mathsf{CR}_2^L)$.

[9] For notational convenience, throughout the section we omit to write the encoding $e : S_N \to \mathbb{Z}_{2^\ell}^N$ and its inverse.

Protocol 8 below provides a protocol $\Pi_{\mathsf{GenBitsPerm}}$ for counting sorting.[10] Here, '$\cdot$' in Step 5a denotes the XOR permutation action. Thanks to the use of GBP-prep, the protocol takes only two communication rounds, and $(2^L-1)\lambda + L$ bits are sent per record and party when we use the XORShuffle protocol.

Protocol 8. $\Pi_{\mathsf{GenBitsPerm}}$

Input: $[\boldsymbol{k}^L]^B, \ldots, [\boldsymbol{k}^1]^B, \gamma \in S_N$.
Output: $[\sigma]$, where σ is the stable sorting of $\boldsymbol{x} \cdot \gamma^{-1} = (\sum_{j=1}^{L} 2^{j-1}\boldsymbol{k}^j) \cdot \gamma^{-1}$.
Subfunctionality: GBP-prep.

(1) The parties call $\mathcal{F}_{\text{GBP-prep}}$ N times parallelly to receive correlated randomness CR^L.
(2) $\boldsymbol{p}^j := \mathsf{Reconst}([\boldsymbol{k}^j]^B \oplus [\boldsymbol{c}^j]^B)$ for $j \in \{1, \ldots, L\}$.
(3) $\boldsymbol{p}^j := \boldsymbol{p}^j \cdot \gamma^{-1}$ for $j \in \{1, \ldots, L\}$.
(4) $\mathsf{CR}^L := \mathsf{CR}^L \cdot \gamma^{-1}$.
(5) For $n \in \{1, \ldots, N\}$ in parallel:
  (a) $[\boldsymbol{w}^{(n)}]^B := [\boldsymbol{v}^{(n)}]^B \cdot g_n$, where $g_n := (\boldsymbol{p}_n^L, \ldots, \boldsymbol{p}_n^1)$
  (b) $[\boldsymbol{f}_h^{(n)}] := \sum_{k=1}^{h} [\boldsymbol{w}_k^{(n)}] + \sum_{l=1}^{n-1} \sum_{k=1}^{N} [\boldsymbol{w}_k^{(l)}]$ for $h \in \{1, \ldots, 2^L\}$
  (c) $[\boldsymbol{m}^{(n)}] \leftarrow \mathsf{XORShuffle}([\boldsymbol{f}^{(n)}]; \langle \boldsymbol{c}_n \rangle)$
  (d) $[\sigma_n] \leftarrow [\boldsymbol{m}_{p_n}^{(n)}]$, where $p_n = 1 + \sum_{i=1}^{L} 2^{i-1}\boldsymbol{p}_n^i$

Correctness is shown as follows: Let $\boldsymbol{k}_n = \sum_{j=1}^{L} 2^{j-1}\boldsymbol{k}_n^j + 1$. Then $\boldsymbol{w}^{(n)}$ is the $\boldsymbol{k}_n$-th unit vector of length 2^L since the following holds:

$$\boldsymbol{w}^{(n)} = \boldsymbol{v}^{(n)} \cdot g_n = \boldsymbol{e}^{(1)} \cdot (c_n \circ g_n) = \boldsymbol{e}^{(1)} \cdot (\boldsymbol{k}_n^L, \ldots, \boldsymbol{k}_n^1) = \boldsymbol{e}^{(\boldsymbol{k}_n)}.$$

On the other hand, by similar arguments as Π_{Mux} (Appendix D.1), Step 5c-5d returns σ_n whose value equals the $\boldsymbol{k}_n$-th element of $\boldsymbol{f}^{(n)}$. Therefore, the correctness of $\Pi_{\mathsf{GenBitsPerm}}$ follows directly from that of the ordinary (plaintext) counting sort algorithm. We have the following theorem.

Theorem 6.1. *The protocol* $\Pi_{\mathsf{GenBitsPerm}}$ *for* GenBitsPerm *is perfectly semi-honest secure in the* GBP-prep*-hybrid model.*

6.3 Optimization 2: Precomputation for Sequential Shuffling

In order to reduce the number of oblivious shuffles, we extend the unshuffling technique. In our setting, double sharings and correlated randomness with respect to any composite of data-independent permutations are precomputable in the offline phase. By using this functionality, we can combine sequential shuffle/unshuffle steps into one step in the online phase. We describe the ideal functionality Radix-Prep for the precomputation for our protocol, which takes a natural number n as a public input. Looking ahead, n is determined by ℓ and L in our protocol. (It is also implicitly parameterized by the number N of elements to which we apply sorting.)

[10] Here, $\mathsf{CR}^L \cdot \gamma^{-1}$ denotes the tuple obtained by permuting each element of CR^L by γ^{-1}, and $\boldsymbol{v}^{(n)}$ denotes the vector $\boldsymbol{v}$ in the n-th CR^L.

Functionality Radix-Prep

Input: $n \in \mathbb{N}$.
Output: $((\langle \pi_k \rangle_1, \langle \pi_k^{-1} \circ \pi_{k+1} \rangle_1), (\langle \pi_k \rangle_2, \langle \pi_k^{-1} \circ \pi_{k+1} \rangle_2))$ for $k \in \{1, \ldots, n\}$, where $\pi_{n+1} := \mathsf{id}$.

(1) Sample $\pi_k \xleftarrow{\$} S_N$ and compute $\langle \pi_k \rangle \leftarrow \mathsf{DShare}_{S_N}(\pi_k)$ for $k \in \{1, \ldots, n\}$.
(2) Compute $\langle \pi_k^{-1} \circ \pi_{k+1} \rangle \leftarrow \mathsf{DShare}_{S_N}(\pi_k^{-1} \circ \pi_{k+1})$ for $k \in \{1, \ldots, n\}$.
(3) Return $((\langle \pi_k \rangle_1, \langle \pi_k^{-1} \circ \pi_{k+1} \rangle_1), (\langle \pi_k \rangle_2, \langle \pi_k^{-1} \circ \pi_{k+1} \rangle_2))$ for $k \in \{1, \ldots, n\}$.

The above functionality for generating correlated randomness is trivial to realize in the client-aided setting. We can also realize it in the standard two-party setting; see the full version for the details.

6.4 Our Radix Sort Protocol

The protocol $\Pi_{\mathsf{RadixSort}}$ is described below, which uses Radix-Prep, GenBitsPerm, and Shuffle as subfunctionalities. We note that if ℓ is not divisible by L, then we will replace L at the last sorting step with the remainder of the division.

The correctness is shown by induction on k. It suffices to show that $\rho_k := \sigma_k \circ \cdots \circ \sigma_1$ is the stable sorting of $\sum_{j=1}^{Lk} 2^{j-1}\boldsymbol{k}^j$ and $\delta_k = \rho_k \circ \pi_k$ for all k, where we define $\pi_1 := \mathsf{id}$ and $\sigma_1 := \delta_1$ is the stable sorting of $\boldsymbol{k}^1$. The base case clearly holds. The induction hypothesis gives

$$\delta_k = \sigma_k \circ \gamma_k = \sigma_k \circ (\delta_{k-1} \circ \pi_{k-1}^{-1} \circ \pi_k) = \rho_k \circ \pi_k$$
$$\boldsymbol{c}^j = \boldsymbol{b}^j \cdot \gamma_k^{-1} = (\boldsymbol{k}^j \cdot \pi_k) \cdot (\delta_{k-1} \circ \pi_{k-1}^{-1} \circ \pi_k)^{-1} = \boldsymbol{k}^j \cdot \rho_{k-1}^{-1}$$

for $j = L(k-1)+1, \ldots, Lk$, which means that σ_k is the stable sorting of $\sum_{j=L(k-1)+1}^{Lk} 2^j(\boldsymbol{k}^j \cdot \rho_{k-1}^{-1})$. By the induction hypothesis, the claim reduces to the correctness of the plaintext L-bit radix sort algorithm in Section 6.1.

Protocol 9. $\Pi_{\mathsf{RadixSort}}$

Input: $[\boldsymbol{k}^\ell]_i^B, \ldots, [\boldsymbol{k}^1]_i^B$ from P_i, where $\boldsymbol{k}^j \in \mathbb{Z}_2^N$ for each $j \in \{1, \ldots, \ell\}$.
Output: $[\rho]_i$ for P_i, where $\rho \in S_N$ is the stable sorting of $\boldsymbol{x} = \sum_{j=1}^{\ell} 2^{j-1}\boldsymbol{k}^j \in \mathbb{Z}_{2^\ell}^N$ and $[\rho] \leftarrow \mathsf{Share}(\rho)$.
Subfunctionality: Radix-Prep, GenBitsPerm, Shuffle.

(1) The parties invoke Radix-Prep$(\lceil \frac{\ell}{L} \rceil - 1)$, and regard the result as $\{\langle \pi_k \rangle\}_{k \in \{2, \ldots \lceil \frac{\ell}{L} \rceil\}}$ and $\{\langle \pi_{k-1}^{-1} \circ \pi_k \rangle\}_{k \in \{3, \ldots, \lceil \frac{\ell}{L} \rceil + 1\}}$, where $\pi_{\lceil \frac{\ell}{L} \rceil + 1} := \mathsf{id}$. Note that $\pi_1 := \mathsf{id}$ so that $\langle \pi_1^{-1} \circ \pi_2 \rangle = \langle \pi_2 \rangle$.
(2) $[\delta_1] \leftarrow \mathsf{GenBitsPerm}([\boldsymbol{k}^L]^B, \ldots, [\boldsymbol{k}^1]^B, \mathsf{id})$.
(3) For $k = 2, 3, \ldots, \lceil \frac{\ell}{L} \rceil$ do:
  (a) $[\boldsymbol{b}^j]^B \leftarrow \mathsf{Shuffle}([\boldsymbol{k}^j]^B; \langle \pi_k \rangle)$ for $j \in \{L(k-1)+1, \ldots, Lk\}$.
  (b) $[\gamma_k] \leftarrow \mathsf{Shuffle}([\delta_{k-1}]; \langle \pi_{k-1}^{-1} \circ \pi_k \rangle)$.
  (c) $\gamma_k := \mathsf{Reconst}([\gamma_k])$.
  (d) $[\sigma_k] \leftarrow \mathsf{GenBitsPerm}([\boldsymbol{b}^{Lk}]^B, \ldots, [\boldsymbol{b}^{L(k-1)+1}]^B, \gamma_k)$.
  (e) $[\delta_k] := [\sigma_k] \cdot \gamma_k$.
(4) $[\rho] \leftarrow \mathsf{Shuffle}([\delta_{\lceil \frac{\ell}{L} \rceil}]; \langle \pi_{\lceil \frac{\ell}{L} \rceil}^{-1} \rangle)$.
(5) Each party P_i outputs $[\rho]_i$.

The security of $\Pi_{\mathsf{RadixSort}}$ is guaranteed as follows.

Theorem 6.2. *The protocol* $\Pi_{\mathsf{RadixSort}}$ *for computing* BooleanStableSort *is perfectly semi-honest secure in the* (Radix-Prep, GenBitsPerm, Shuffle)*-hybrid model.*

7 PERFORMANCE EVALUATIONS

We evaluate performances of our main applications, oblivious shuffle and oblivious sort, and compare to the state-of-the-arts, namely Chase *et al.* [9] (CGP) and Chida *et al.* [11] (CHI+), respectively. These are summarized in Tables 1 and 2. The number of parties indicates the setting of each scheme. In particular, in the client-aided setting (for scheme c, e, h in Table 1,2), we have three parties in the offline phase and two in the online phase. All the other settings are the standard two-party or three-party settings. As for security, all the schemes shown here are semi-honest secure that allows a corruption of one party. The original CGP protocol (scheme a in Table 1) does not consider the offline/online paradigm. In Appendix B.1, we modify their scheme to accommodate such a setting (scheme b), and further utilize the client-aided method (scheme c).

We derive the communication and computation times based on counting argument, in the same vein as in [9]. The analysis for each value in the table is given in Appendix B (see Table 1,2 for detailed section information). Note that we optimize the offline phase for the schemes in the client-aided setting (for scheme c, e, h, i in Table 1,2) in their respective appendix.

Comparing Shuffle Protocols. A core improvement of our protocols (scheme d,e) is the online time: our protocols are two times faster than the *modified* CGP schemes (scheme b,c), which are firstly considered in our paper. To see the "overall" improvement of our fastest shuffle protocol (scheme e), which is in the client-aided setting, compared to the *original* CGP shuffle protocol (scheme a), which is in the standard two-party setting, ours is much faster as the dominant term T_1 in the CGP protocol could be much larger than the total time of our scheme e. This improvement captures both the advantages of exploiting client-aided computation and the structural improvements of our approach.

For concrete comparison, we plot values from Table 1 in Fig. 3,4,5. We defer most plots to the appendix, but include two plots in Fig. 1 and summarize in Table 3. We show the total time for ours (scheme e) versus the original CGP (scheme a) where ℓ range from $\ell = 64$ to $\ell = 640,000$, for $N = 2^{20}$ and $N = 2^{32}$, with three different network speeds ($B = 50Mbps$ for the WAN setting, and $B = 1Gbps$ or 10Gbps for the LAN setting), in Fig. 1,3. As shown, our client-aided protocol is up to around 152 times faster than the original CGP protocol (which is in the standard two-party setting) in the WAN setting, and up to around 105 times faster in the LAN setting.

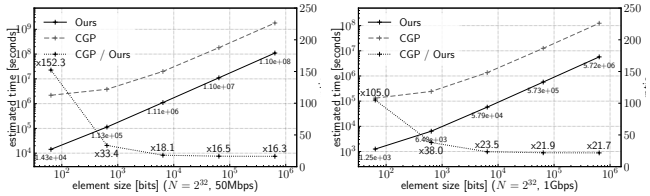

Figure 1: Plotting *total* running times for our *shuffle* (scheme e), compared to the original CGP (scheme a)

Of all the scenarios we considered, the lowest speed advantage of our protocol is 10.2 times faster than the original CGP protocol.

As shown in Table 3, when comparing the *client-aided* schemes, namely, the client-aided modified CGP protocol (scheme c) with our client-aided protocol (scheme e), the total time is also improved about a factor of 1.34 on average. On the other hand, comparing the *standard two-party* schemes, namely, the modified CGP protocol (scheme b) with our shuffle protocol with pre-processing protocols (scheme d), the total time is about 0.50 times slower. Comparing to the *standard three-party* protocol in [11] (scheme f), our client-aided protocol (scheme e) is slightly slower in total time. However, we emphasize that our key improvement is the online time, which is two times faster, in all comparisons. In App. B.4, we also implement the full shuffle protocols (with $\ell = 64$) to confirm our estimated running times.

Comparing Sort Protocols. In Table 2, we compare our client-aided sort protocols with parameter $L = 2, 3$ (scheme h, scheme i, resp.) to the three-party protocol of Chida *et al.* [11] (scheme g), choosing the parameters $L = 2, 3$ and $\lambda = 32$ (the analysis is deferred to the appendix). As shown, we improve the number of online rounds by more than 2.6 times (comparing scheme i to scheme g). Our protocol also improves the online running times compared to that of [11]. For concreteness, we plot our online running time compared to those of the three-party protocol of [11] with $\ell = 64$, while $t_{\mathrm{lat}} = 0.02$ and 0 seconds, in the WAN, LAN setting, respectively, in Figure 6,7 in App. B.5, and show some of these results in Figure 2 below. As shown, for the *online* time (which is our main focus), our protocol is 1.22 to 2.57 times faster than that of [11] in all the scenarios we considered here. For the *total* time, ours is faster for a smaller number of elements, up to $N \approx 20000$, in the WAN setting; this leaves room for improvement on the total running time for larger N or the LAN setting.

Besides the client-aided setting, we also consider pre-processing protocols (see the full version for the details). This yields a standard two-party sort protocol (without client-aided computation). Note that the online time (of the overall protocol) is the same as scheme h and i, respectively for $L = 2, 3$.

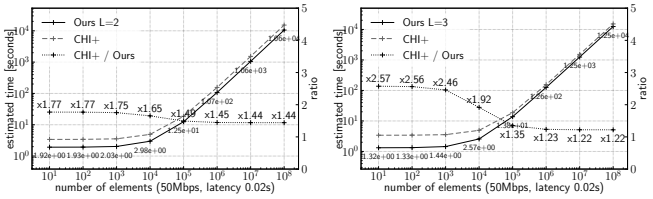

Figure 2: *Online* running times for our *oblivious sort* protocols (scheme h,i resp.), compared to CHI+ (scheme g)

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. This work is partially supported by JST, CREST Grant Number JPMJCR19F6, and JSPS, KAKENHI Grant Number 19H01109, Japan.

Table 1: Comparison for Shuffle Protocols

	Scheme	Parties		Online	Communication time		Computation time	
		offline	online	round	offline	online	offline	online
a	CGP Original [9] (§B.1)	-	2	6	-	T_1	-	$2T_2 + t_{\text{perm}}$
b	CGP Modified (§A,§B.2)	2	2	2	T_1	$2\frac{\ell N}{B}$	$2T_2 + t_{\text{perm}}$	negl
c	CGP Modified + Client-aided (§A,§B.2)	3	2	2	$\frac{\ell N}{B}$	$2\frac{\ell N}{B}$	$3T_3 + 2t_{\text{perm}}$	negl
d	Ours w/Offline protocols (§B.3)	2	2	1	$2T_1 - \frac{(\ell \log N - 64)N}{2B}$	$\frac{\ell N}{B}$	$4T_2 + 2t_{\text{perm}}$	negl
e	Ours w/Client-aided (§3.5,§B.3)	3	2	1	$\frac{(\ell+32)N}{B}$	$\frac{\ell N}{B}$	$3T_3 + 3t_{\text{perm}}$	negl
f	CHI+ Shuffle [11] (§B.6)	3	3	2	negl	$2\frac{\ell N}{B}$	$2T_3 + 2t_{\text{perm}}$	negl

Note: N is the number of elements, ℓ is the bit length of each element, t_{AES} is the time of one AES execution, B is the network speed, and t_{perm} is the time for generating a random permutation (depended on N). Denote $T_1 = \frac{\ell N \log N + 768(\log N - 2)}{B}$, $T_2 = N(\log N - 2)(22 + 8\frac{\ell}{k})t_{\text{AES}}$ (these are deduced from [9], see B.1), and $T_3 = \frac{N\ell t_{\text{AES}}}{128}$ (time for generating a random vector), negl refers to negligible. All are semi-honest secure with one corruption.

Table 2: Comparison for Sort Protocols

	Scheme	Parties		Online	Communication time		Computation time	
		offline	online	round	offline	online	offline	online
g	CHI+ Sort [11] (§B.6)	3	3	$8\lceil\frac{\ell}{3}\rceil - 4$	$\lceil\frac{\ell}{3}\rceil\frac{192N}{B}$	$\lceil\frac{\ell}{3}\rceil(\frac{352N}{B} + 8t_{\text{lat}})$	$2\lceil\frac{\ell}{3}\rceil t_{\text{perm}}$	negl
h	Ours w/Client-aided, $L = 2$ (§6,§B.5)	3	2	$3\lceil\frac{\ell}{2}\rceil$	$\lceil\frac{\ell}{2}\rceil\frac{352N}{B}$	$\lceil\frac{\ell}{2}\rceil(\frac{160N}{B} + 3t_{\text{lat}})$	$5\lceil\frac{\ell}{2}\rceil t_{\text{perm}}$	negl
i	Ours w/Client-aided, $L = 3$ (§6,§B.5)	3	2	$3\lceil\frac{\ell}{3}\rceil$	$\lceil\frac{\ell}{3}\rceil\frac{576N}{B}$	$\lceil\frac{\ell}{3}\rceil(\frac{288N}{B} + 3t_{\text{lat}})$	$5\lceil\frac{\ell}{3}\rceil t_{\text{perm}}$	negl

Note: t_{lat} is the latency of one communication round (and refer to other parameters from Table 1). All are semi-honest secure with one corruption.

Table 3: Numerical Improvements

	Scheme pair	Total speed	Online speed	reference
Shuffle	a → e	10×-105×*	-	Fig. 1,3
		13×-152×**	-	Fig. 1,3
	c → e	1.34×†	2×	Fig. 4,5
	b → d	> 0.50×	2×	
	f → e	0.91×†	2×	Fig. 8
Sort	g → h, i	> 1×‡	1.22-2.57 ×	Fig. 6,7

Note: 2× means two times faster. (*) for the LAN case. (**) for the WAN case. (†) on average. (‡) in the case of a smaller number of elements, up to $N \approx 20000$, in the WAN setting (other cases in this line attain < 1 ×).

REFERENCES

[1] Donald Beaver. 1992. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO'91 (LNCS, Vol. 576)*, Joan Feigenbaum (Ed.). Springer, Heidelberg, 420–432. https://doi.org/10.1007/3-540-46766-1_34

[2] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *20th ACM STOC*. ACM Press, 1–10. https://doi.org/10.1145/62212.62213

[3] Dan Bogdanov, Sven Laur, and Riivo Talviste. 2013. *Oblivious Sorting of Secret-Shared Data*. Technical Report T-4-19. Cybernetica. http://research.cyber.ee/

[4] Dan Bogdanov, Sven Laur, and Riivo Talviste. 2014. A Practical Analysis of Oblivious Sorting Algorithms for Secure Multi-party Computation. In *NordSec 2014 (LNCS, Vol. 8788)*, Karin Bernsmed and Simone Fischer-Hübner (Eds.). Springer, Heidelberg, 59–74. https://doi.org/10.1007/978-3-319-11599-3_4

[5] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. 2020. Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation. Cryptology ePrint Archive, Report 2020/1392. https://eprint.iacr.org/2020/1392.

[6] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function Secret Sharing. In *EUROCRYPT 2015, Part II (LNCS, Vol. 9057)*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer, Heidelberg, 337–367. https://doi.org/10.1007/978-3-662-46803-6_12

[7] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2019. Secure Computation with Preprocessing via Function Secret Sharing. In *TCC 2019, Part I (LNCS, Vol. 11891)*, Dennis Hofheinz and Alon Rosen (Eds.). Springer, Heidelberg, 341–371. https://doi.org/10.1007/978-3-030-36030-6_14

[8] Octavian Catrina and Sebastiaan de Hoogh. 2010. Improved Primitives for Secure Multiparty Integer Computation. In *SCN 10 (LNCS, Vol. 6280)*, Juan A. Garay and Roberto De Prisco (Eds.). Springer, Heidelberg, 182–199. https://doi.org/10.1007/978-3-642-15317-4_13

[9] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. 2020. Secret-Shared Shuffle. In *ASIACRYPT 2020, Part III (LNCS, Vol. 12493)*, Shiho Moriai and Huaxiong Wang (Eds.). Springer, Heidelberg, 342–372. https://doi.org/10.1007/978-3-030-64840-4_12

[10] Huajie Chen, Ali Burak Ünal, Mete Akgün, and Nico Pfeifer. 2020. Privacy-Preserving SVM on Outsourced Genomic Data via Secure Multi-Party Computation. In *IWSPA 20*, Rakesh Verma, Latifur Khan, and Chilukuri K. Mohan (Eds.). ACM Press, 61–69. https://doi.org/10.1145/3375708.3380316

[11] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Naoto Kiribuchi, and Benny Pinkas. 2019. An Efficient Secure Three-Party Sorting Protocol with an Honest Majority. Cryptology ePrint Archive, Report 2019/695. https://eprint.iacr.org/2019/695.

[12] Michele Ciampi and Claudio Orlandi. 2018. Combining Private Set-Intersection with Secure Two-Party Computation. In *SCN 18 (LNCS, Vol. 11035)*, Dario Catalano and Roberto De Prisco (Eds.). Springer, Heidelberg, 464–482. https://doi.org/10.1007/978-3-319-98113-0_25

[13] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. 2006. Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation. In *TCC 2006 (LNCS, Vol. 3876)*, Shai Halevi and Tal Rabin (Eds.). Springer, Heidelberg, 285–304. https://doi.org/10.1007/11681878_15

[14] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS 2015*. The Internet Society.

[15] David S. Dummit and Richard M. Foote. 2004. *Abstract algebra* (3rd ed.). Wiley.

[16] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *19th ACM STOC*, Alfred Aho (Ed.). ACM Press, 218–229. https://doi.org/10.1145/28395.28420

[17] Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2014. Oblivious Radix Sort: An Efficient Sorting Algorithm for Practical Secure Multi-party Computation. Cryptology ePrint Archive, Report 2014/121. https://eprint.iacr.org/2014/121.

[18] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2013. Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms. In *ICISC 2012 (LNCS, Vol. 7839)*, Taekyoung Kwon, Mun-Kyu Lee, and

Daesung Kwon (Eds.). Springer, Heidelberg, 202–216. https://doi.org/10.1007/978-3-642-37682-5_15
[19] Carmit Hazay. 2015. Oblivious Polynomial Evaluation and Secure Set-Intersection from Algebraic PRFs. In *TCC 2015, Part II (LNCS, Vol. 9015)*, Yevgeniy Dodis and Jesper Buus Nielsen (Eds.). Springer, Heidelberg, 90–120. https://doi.org/10.1007/978-3-662-46497-7_4
[20] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. 2012. Secure two-party computations in ANSI C. In *ACM CCS 2012*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM Press, 772–783. https://doi.org/10.1145/2382196.2382278
[21] Yan Huang, David Evans, and Jonathan Katz. 2012. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?. In *NDSS 2012*. The Internet Society.
[22] Kristján Valur Jónsson, Gunnar Kreitz, and Misbah Uddin. 2011. Secure Multi-Party Sorting and Applications. Cryptology ePrint Archive, Report 2011/122. https://eprint.iacr.org/2011/122.
[23] Seny Kamara, Payman Mohassel, and Ben Riva. 2012. Salus: a system for server-aided secure function evaluation. In *ACM CCS 2012*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM Press, 797–808. https://doi.org/10.1145/2382196.2382280
[24] Ryo Kikuchi, Dai Ikarashi, Takahiro Matsuda, Koki Hamada, and Koji Chida. 2018. Efficient Bit-Decomposition and Modulus-Conversion Protocols with an Honest Majority. In *ACISP 18 (LNCS, Vol. 10946)*, Willy Susilo and Guomin Yang (Eds.). Springer, Heidelberg, 64–82. https://doi.org/10.1007/978-3-319-93638-3_5
[25] Peeter Laud. 2015. A Private Lookup Protocol with Low Online Complexity for Secure Multiparty Computation. In *ICICS 14 (LNCS, Vol. 8958)*, Lucas Chi Kwong Hui, S. H. Qing, Elaine Shi, and S. M. Yiu (Eds.). Springer, Heidelberg, 143–157. https://doi.org/10.1007/978-3-319-21966-0_11
[26] Peeter Laud and Martin Pettai. 2016. Secure Multiparty Sorting Protocols with Covert Privacy. In *NordSec 2016 (LNCS, Vol. 10014)*, Billy Bob Brumley and Juha Röning (Eds.). Springer, Heidelberg, 216–231. https://doi.org/10.1007/978-3-319-47560-8_14
[27] Sven Laur, Riivo Talviste, and Jan Willemson. 2013. From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In *ACNS 13 (LNCS, Vol. 7954)*, Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini (Eds.). Springer, Heidelberg, 84–101. https://doi.org/10.1007/978-3-642-38980-1_6
[28] Sven Laur, Jan Willemson, and Bingsheng Zhang. 2011. Round-Efficient Oblivious Database Manipulation. In *ISC 2011 (LNCS, Vol. 7001)*, Xuejia Lai, Jianying Zhou, and Hui Li (Eds.). Springer, Heidelberg, 262–277. https://doi.org/10.1007/978-3-642-24861-0_18
[29] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *ACM CCS 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, 35–52. https://doi.org/10.1145/3243734.3243760
[30] Payman Mohassel, Peter Rindal, and Mike Rosulek. 2019. Fast Database Joins for Secret Shared Data. Cryptology ePrint Archive, Report 2019/518. https://eprint.iacr.org/2019/518.
[31] Payman Mohassel and Seyed Saeed Sadeghian. 2013. How to Hide Circuits in MPC an Efficient Framework for Private Function Evaluation. In *EUROCRYPT 2013 (LNCS, Vol. 7881)*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer, Heidelberg, 557–574. https://doi.org/10.1007/978-3-642-38348-9_33
[32] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 19–38. https://doi.org/10.1109/SP.2017.12
[33] Hiraku Morita, Nuttapong Attrapadung, Tadanori Teruya, Satsuya Ohata, Koji Nuida, and Goichiro Hanaoka. 2018. Constant-Round Client-Aided Secure Comparison Protocol. In *ESORICS 2018, Part II (LNCS, Vol. 11099)*, Javier López, Jianying Zhou, and Miguel Soriano (Eds.). Springer, Heidelberg, 395–415. https://doi.org/10.1007/978-3-319-98989-1_20
[34] Mahnush Movahedi, Jared Saia, and Mahdi Zamani. 2015. Secure Multi-party Shuffling. In *SIROCCO 2015 (LNCS, Vol. 9439)*, Christian Scheideler (Ed.). Springer, Heidelberg, 459–473. https://doi.org/10.1007/978-3-319-25258-2_32
[35] Moni Naor and Benny Pinkas. 2006. Oblivious Polynomial Evaluation. *SIAM J. Comput.* 35, 5 (2006), 1254–1281. https://doi.org/10.1137/S0097539704383633
[36] Takashi Nishide and Kazuo Ohta. 2007. Multiparty Computation for Interval, Equality, and Comparison Without Bit-Decomposition Protocol. In *PKC 2007 (LNCS, Vol. 4450)*, Tatsuaki Okamoto and Xiaoyun Wang (Eds.). Springer, Heidelberg, 343–360. https://doi.org/10.1007/978-3-540-71677-8_23
[37] Satsuya Ohata and Koji Nuida. 2020. Communication-Efficient (Client-Aided) Secure Two-Party Protocols and Its Application. In *FC 2020 (LNCS, Vol. 12059)*, Joseph Bonneau and Nadia Heninger (Eds.). Springer, Heidelberg, 369–385. https://doi.org/10.1007/978-3-030-51280-4_20
[38] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. 2019. Efficient Circuit-Based PSI with Linear Communication. In *EUROCRYPT 2019, Part III (LNCS, Vol. 11478)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, Heidelberg, 122–153. https://doi.org/10.1007/978-3-030-17659-4_5
[39] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. 2018. Efficient Circuit-Based PSI via Cuckoo Hashing. In *EUROCRYPT 2018, Part III (LNCS, Vol. 10822)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer, Heidelberg, 125–157. https://doi.org/10.1007/978-3-319-78372-7_5
[40] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *ASIACCS 18*, Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim (Eds.). ACM Press, 707–721. https://doi.org/10.1145/3196494.3196522
[41] Nigel P. Smart and Titouan Tanguy. 2019. TaaS: Commodity MPC via Triples-as-a-Service. In *CCSW 19*, Radu Sion and Charalampos Papamanthou (Eds.). ACM Press, 105–116. https://doi.org/10.1145/3338466.3358918
[42] Vishnu Subramanian. 2018. *Deep Learning with PyTorch*. Packt.
[43] Riivo Talviste. 2016. *Applying Secure Multi-Party Computation in Practice*. Ph. D. Dissertation. Univ. of Tartu.
[44] Guan Wang, Tongbo Luo, Michael T. Goodrich, Wenliang Du, and Zutao Zhu. 2010. Bureaucratic protocols for secure two-party sorting, selection, and permuting. In *ASIACCS 10*, Dengguo Feng, David A. Basin, and Peng Liu (Eds.). ACM Press, 226–237. https://doi.org/10.1145/1755688.1755716
[45] Lei Wei and Michael K. Reiter. 2012. Third-Party Private DFA Evaluation on Encrypted Files in the Cloud. In *ESORICS 2012 (LNCS, Vol. 7459)*, Sara Foresti, Moti Yung, and Fabio Martinelli (Eds.). Springer, Heidelberg, 523–540. https://doi.org/10.1007/978-3-642-33167-1_30
[46] Andrew Chi-Chih Yao. 1982. Protocols for Secure Computations (Extended Abstract). In *23rd FOCS*. IEEE Computer Society Press, 160–164. https://doi.org/10.1109/SFCS.1982.38
[47] Bingsheng Zhang. 2011. Generic Constant-Round Oblivious Sorting Algorithm for MPC. In *ProvSec 2011 (LNCS, Vol. 6980)*, Xavier Boyen and Xiaofeng Chen (Eds.). Springer, Heidelberg, 240–256. https://doi.org/10.1007/978-3-642-24316-5_17
[48] Yihua Zhang, Aaron Steele, and Marina Blanton. 2013. PICCO: a general-purpose compiler for private distributed computation. In *ACM CCS 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, 813–826. https://doi.org/10.1145/2508859.2516752

A REVISITING THE CGP SHUFFLE

In this section, we revisit the oblivious shuffle protocol by Chase, Ghosh, and Poburinnaya (CGP) [9]. We first briefly review their scheme. We then modify it into the offline/online paradigm, with some optimizations along the way. We will argue that this modified CGP protocol inherently requires *two rounds* in the online phase (confirming an argument intuitively described in Section 1.1). Finally, we consider the modified CGP protocol in the client-aided setting, and compare to our shuffle protocol (in the same setting).

Brief Review of the CGP protocol We first briefly review the shuffle protocol of Chase *et al.* [9]. Their fundamental protocol for oblivious shuffling is Permute+Share, where parties inputs are $(\pi \in S_n, \boldsymbol{v} \in R^n)$ and outputs $[\boldsymbol{v} \cdot \pi]$. Here, we use our terminology and instantiate the (right) group action $\cdot$ as the place-permutation. In the protocol Permute+Share, parties first execute the subprotocol ShareTrans, where parties inputs are $(\pi, \perp)$ and their outputs are $(\boldsymbol{a} \cdot \pi + \boldsymbol{c}, \{\boldsymbol{a}, \boldsymbol{c}\})$ for random $\boldsymbol{a}, \boldsymbol{c} \in R^n$. After the execution of ShareTrans, the protocol Permute+Share proceeds as follows:

(1) P_2 locally computes $\boldsymbol{z} = \boldsymbol{v} + \boldsymbol{a}$ and sends it to P_1.
(2) P_1 outputs $[y]_1 := \boldsymbol{z} \cdot \pi - (\boldsymbol{a} \cdot \pi + \boldsymbol{c})$, and P_2 outputs $[y]_2 := \boldsymbol{c}$.

The shuffle protocol of Chase *et al.* is obtained by two sequential executions of Permute+Share with two random permutation inputs. Each party holds a correlated randomness generated by ShareTrans, which consists of one permutation and three vectors.

Making CGP Offline/Online We are now ready to modify their scheme into the offline/online setting. We quickly observe that each permutation input π of Permute+Share does not necessarily have to be chosen during execution, so that each first execution of ShareTrans can be seen as a data-independent computation and can be carried out in the offline phase. Next, we also observe that

each input π does not necessarily have to be chosen by each party. Instead, we capture the correlated randomness in ShareTrans in the functionality ModifiedCGP_prep, described below. By performing the two executions of ShareTrans consecutively and collectively, and by combining several terms, we can reduce the elements of a correlated randomness for each party to one permutation and two vectors.

Functionality ModifiedCGP_prep

(1) Sample $\pi_1, \pi_2 \xleftarrow{\$} S_N$ and $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \xleftarrow{\$} R^N$.
(2) Compute $\boldsymbol{\alpha} = \boldsymbol{b} \cdot \pi_1 + \boldsymbol{c}$ and $\boldsymbol{\beta} = \boldsymbol{c} \cdot \pi_2 + \boldsymbol{a}$.
(3) Return $((\pi_1, \boldsymbol{a}, \boldsymbol{\alpha}), (\pi_2, \boldsymbol{b}, \boldsymbol{\beta}))$.

We now obtain the shuffle protocol $\Pi_{\mathsf{ModifiedCGP}}$, as described below. As mentioned above, we carry out Step 1 of the protocol in the offline phase. In the online phase the protocol takes two rounds, and in each round one party needs to send n elements of R to the other party. In other words, we require the communication time in the online phase for sending $2n$ elements in total.

Protocol 10. $\Pi_{\mathsf{ModifiedCGP}}$

Input: $[\boldsymbol{x}]_i$ from P_i, where $\boldsymbol{x} \in R^N$.
Output: $([\boldsymbol{y}]_i, \pi_i)$ for P_i, where $\pi_i \xleftarrow{\$} S_n$, $\boldsymbol{y} = \boldsymbol{x} \cdot \pi$ for $\pi = \pi_1 \circ \pi_2$.
Subfunctionality: ModifiedCGP_prep.

(1) Run $((\pi_1, \boldsymbol{a}, \boldsymbol{\alpha}), (\pi_2, \boldsymbol{b}, \boldsymbol{\beta})) \leftarrow$ ModifiedCGP_prep.
(2) P_2 computes $\boldsymbol{w} = [\boldsymbol{x}]_2 + \boldsymbol{b}$, and sends it to P_1.
(3) P_1 computes $\boldsymbol{v} = [\boldsymbol{x}]_1 \cdot \pi_1 + \boldsymbol{w} \cdot \pi_1 - \boldsymbol{\alpha}$ and sends it to P_2.
(4) P_1 outputs $[\boldsymbol{y}]_1 = -\boldsymbol{a}$, and P_2 outputs $[\boldsymbol{y}]_2 = \boldsymbol{v} \cdot \pi_2 + \boldsymbol{\beta}$.

The correctness is shown as follows:

$$\begin{aligned}
[y_i]_2 &= v_{\pi_2(i)} + \beta_i \\
&= ([x_{\pi(i)}]_1 + w_{\pi(i)} - \alpha_{\pi_2(i)}) + \beta_i \\
&= [x_{\pi(i)}]_1 + ([x_{\pi(i)}]_2 + b_{\pi(i)}) - (b_{\pi(i)} + c_{\pi_2(i)}) + (c_{\pi_2(i)} + a_i) \\
&= x_{\pi(i)} + b_{\pi(i)} - b_{\pi(i)} - c_{\pi_2(i)} + c_{\pi_2(i)} + a_i \\
&= x_{\pi(i)} + a_i
\end{aligned}$$

where $\pi = \pi_1 \circ \pi_2$. Thus, it holds that $[\boldsymbol{y}]_1 + [\boldsymbol{y}]_2 = \boldsymbol{x} \cdot \pi$.

The security of $\Pi_{\mathsf{ModifiedCGP}}$ is guaranteed as follows.

Theorem A.1. *The protocol $\Pi_{ModifiedCGP}$ for* Shuffle *is perfectly semi-honest secure in the ModifiedCGP_prep-hybrid model.*

Intuitive Comparison to Our Protocol We now compare the above modified CGP protocol to our main protocol in Section 3.

We first observe that the correlated randomness generated by ModifiedCGP_prep is *asymmetric* in the sense that the output of the parties from ModifiedCGP_prep is not exchangeable. In contrast, using a double-sharing of a group element enables each party to hold a share in a symmetrical fashion, and we can make the correlated randomness *symmetric*.

Next, we can view our protocol Π_{LGA} in Section 3.4 as being constructed by "parallelizing" the two communications in the protocol $\Pi_{\mathsf{ModifiedCGP}}$. In both protocols, the parties locally compute the group action in order, depending on the group element given as a sharing of a group element. The difference between the two protocols is the symmetry of a sharing of group elements. The first communication round of $\Pi_{\mathsf{ModifiedCGP}}$ is only for P_1's local computation of the first group action. This is because the *asymmetry* of a sharing of the group element forces the parties to match the order of $[x]_1$ and $[x]_2$ at first. In the protocol Π_{LGA}, by using a double-sharing and symmetric correlated randomness, we can deal with the order of $[x]_1$ and $[x]_2$ independently, and we can omit the communication before the first local computation of group action.

B DEFERRED ANALYSIS FOR EVALUATIONS

This section provides detailed analysis on evaluations of protocols, deferred from Section 7.

We simulate the running time by estimation based on counting argument, in the same vein as [9]. We calculate the online communication time as $t_{\text{on\_comm}} = (\#\text{ online comm. bits}) \times N/B + (\#\text{ rounds}) \times (\text{latency})$, where N is the number of records. We vary the network speed B as 50 Mbits/sec (WAN setting), 1 and 10 Gbits/sec (LAN settings). We assume full duplex networks, which allow simultaneous sending and receiving, whenever is possible. We also assume that each party connects through a network switch. We set the latency to 0.02 seconds for the WAN setting (and 0 for the LAN settings).[11] Similarly, the offline communication time is $t_{\text{off\_comm}} = (\#\text{ offline comm. bits}) \times N/B$. The local computation is considered to be dominated by the local generation of random permutations and random masks. Let t_{perm} be the generation time for single random permutation and t_{AES} be the running time of one AES execution. We then use the value $t_{\text{AES}} = 3.5$ nanoseconds, as in [9]. We implement the algorithm for generating uniformly random permutations using the method in [28]; for $N = 2^{20}$ and $N = 2^{32}$, t_{perm} is 0.017 and 181.037 seconds, respectively.

Optimization for offline phase in Client-aided setting. In the client-aided setting with computational security, we can use techniques based on a pseudorandom generator (PRG) to reduce the communication for sending correlated randomness in the offline phase, as suggested in, *e.g.,* [40]. In particular, applying the technique to the offline phase of our protocols reduces the data sent by the client as follows:

- In Shuffle-prep, a_1, a_2, a_2' may be generated using PRG. In addition, the method in [28] can be used to locally generate π_1, π_2, π_2'. The client computes a_1', π_1' from them and sends it to P_1.
- In GBP-prep, all elements of CR^L except for $\boldsymbol{v}_1$ and $\boldsymbol{a}_1'$ can be generated by PRG.

B.1 Recapped Analysis for the Chase *et al.* (CGP) Protocol

This subsection recapitulates the analysis of running time for the Chase *et al.* (CGP) shuffle protocol [9]. The scheme of [9] uses some internal parameters k, T, d, where $k = 128, T \le N, d = 2\lceil \log N / \log T \rceil - 1$. Using exactly the same analysis as in [9], we can deduce that the computation time is $2(3dN \log T + dNT(2 + \ell/k))t_{\text{AES}} + t_{\text{perm}}$. This follows from the fact that their Shuffle protocol consists of two Permute+Share protocols that runs dN/T Share Translation protocols of size T. Share Translation protocol requires $3T \log T$ AES calls for OT protocols and $T^2(2 + \ell/k)$ AES calls for local computations. Note that, the estimation in [9] only considered the building block, namely, their Permute+Share protocol; here we estimate for their full protocol that runs Permute+Share twice.

[11] For the sorting protocols, we include latency in estimating communication time, similarly to [11]. For the shuffle protocols, we did not include this since the number of rounds were small and hence overall latency was negligible.

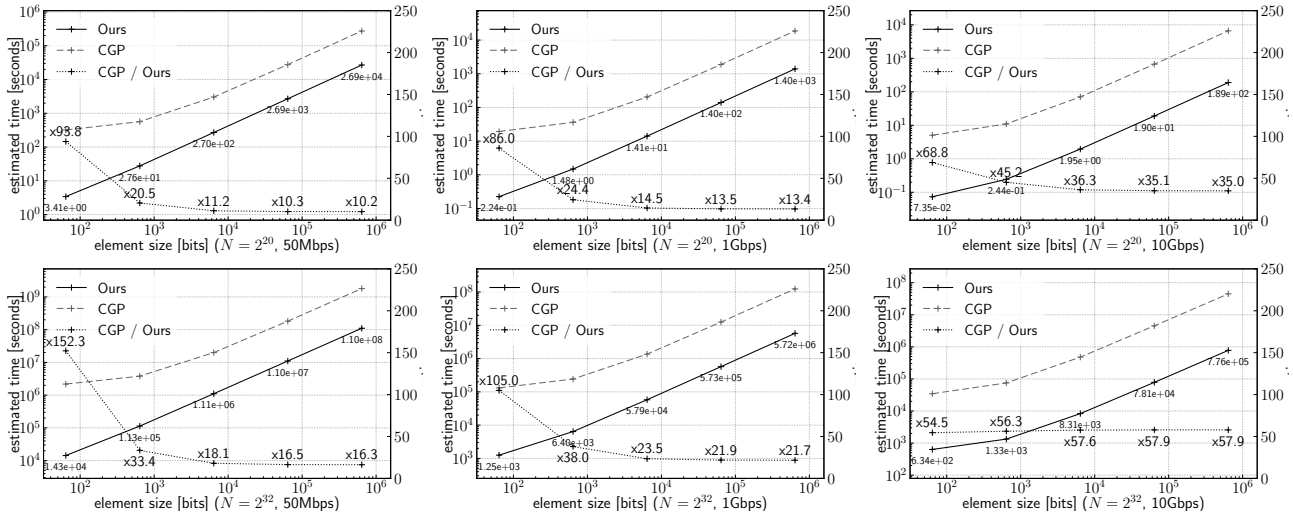

Figure 3: *Total* running times for our *oblivious shuffle* protocols, compared to Chase *et al.* (CGP [9]), with improvement ratios

The communication time is $(3dNk \log T + 2(d+1)N\ell)/B$, where $3dNk \log T$ corresponds to the communication complexity for one Permute+Share and $2(d+1)N\ell$ corresponds to share sendings. Note that the estimation of the communication time is not simply twice as much as that of Permute+Share as we utilize the full duplex network in one simultaneous sending.

Their performances are at best when $T = 16$, and we use this parameter when comparing with ours in Figure 3. To summarize, the communication time is $T_1 = \frac{\ell N \log N + 768(\log N - 2)}{B}$ seconds, and computation time is $2T_2 + t_{\text{perm}}$ where $T_2 = N(\log N - 2)(22 + 8\frac{\ell}{k})t_{\text{AES}}$ seconds.

B.2 Detailed Analysis for Modified CGP Shuffle Protocol

The *Client-aided* Modified CGP protocol. We consider the modified CGP protocol in the client-aided setting, where the correlated randomness is generated by a semi-honest client. By using the pseudorandom secret sharing technique, we can deduce its running time as follows: The offline communication time corresponds to sending one random mask to one server from the client, and is given by $t_{\text{off\_comm}} = N\ell/B$. The offline computation time is dominated by the local generation of 2 permutations and 3 random masks. In the online phase, each server sends $N\ell$ bits sequentially, thus $2N\ell$ bits are sent in total. Local computation times by servers are small and thus negligible. Hence, the online communication time is $2N\ell/B$.

The *2-party* Modified CGP protocol with pre-processing protocols. Share Translation protocol can opt out of the online phase since it is independent from servers' inputs. Note that we realise Share Translation in offline phase by combining Permute+Share and a share vector sending. It may sound redundunt because Permute+Share is already made of Share Translation and we are implementing Share Translation from Permute+Share. This is a security reason; Share Translation has at most permutation hiding property while Permute+Share has static semi-honest security. Therefore, our construction runs Permute+Share as well as a share vector sending in the offline phase. Thus, the offline communication of the modified CGP protocol is as same as the online communication of the original CGP, namely, T_1 seconds.

Comparing the Client-aided Modified CGP to Ours. Figure 4 and 5 show the online and total running time of the client-aided modified CGP shuffle protocol, respectively, compared to our shuffle protocol. For the online running time, ours is two times faster in all cases. For the total running time, ours is up to 1.5 times faster and around 1.34 times faster on average (the average is computed over all the cases we consider here). The client-aided modified CGP protocol is a little bit faster than ours for only the cases of $\ell = 10^2$ in the LAN setting with 10Gbps network speed; these comprise only two cases (plots) out of all the 30 cases (plots) we consider here in Figure 5.

Comparing the Client-aided Modified CGP to the Original CGP. We finally remark that leveraging the original CGP protocol to the client-aided setting already dramatically improves the original CGP protocol. From Figure 3 and Figure 5, we can deduce that the client-aided modified CGP protocol, which is not trivial in the first place and which we proposed in this section, is already about up to $152/1.4 = 108$ times faster than the original CGP protocol in total running time (in the case where $\ell = 64$, $N = 2^{32}$, and B is 50 Mbps). We also view this as a side contribution in this paper.

B.3 Detailed Analysis for Our Shuffle Protocol

Our Shuffle protocol in *Client-aided* setting. For our shuffle protocol, the offline communication time corresponds to sending the correlated randomness to servers, and is given by $t_{\text{off\_comm}} = (\ell + 32)N/B$ (the initializer sends a permutation of 32-bit indices for an N-dimensional vector, and a N-dimensional random vector of ℓ-bit shares). In the online phase, the parties simultaneously send $N\ell$ bits (an N-dimensional vector of ℓ-bit shares, which is locally permuted and masked by the randomness given by the initializer) to each other, and hence the online communication time

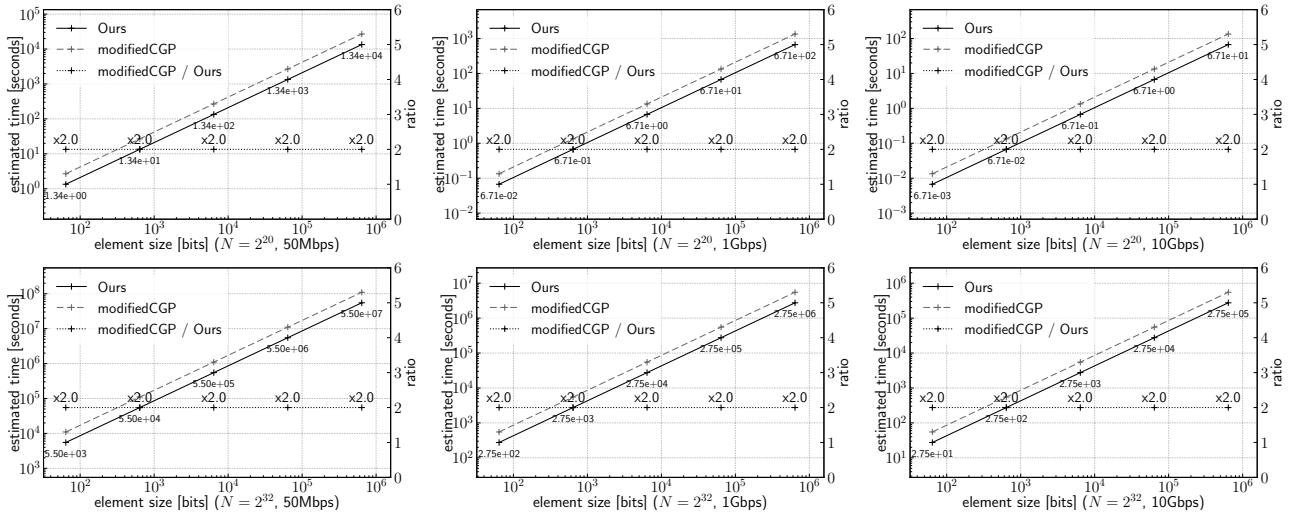

Figure 4: *Online* running times for our *oblivious shuffle* protocols, compared to the client-aided modified CGP protocol, with improvement ratios

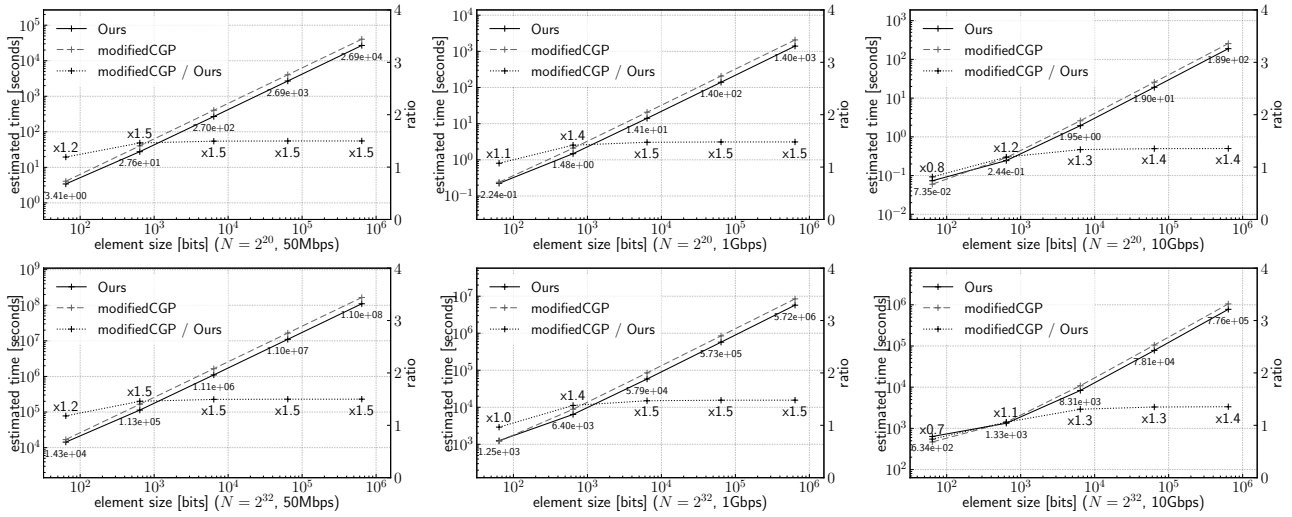

Figure 5: *Total* running times for our *oblivious shuffle* protocols, compared to the client-aided modified CGP protocol, with improvement ratios

Table 4: Benchmarking for our shuffle protocols

Protocol	Part	$N = 2^{18}$	$N = 2^{20}$	$N = 2^{22}$
Shuffle	Offline	0.0212[s]	0.146[s]	0.751[s]
	Online	0.00200[s]	0.0146[s]	0.115[s]
	Total	0.0232[s]	0.161[s]	0.866[s]

is $t_{\text{on\_comm}} = N\ell/B$ (due to the full duplex network). The local computation time is $3t_{\text{perm}}$ (as we generates 3 permutations) and $3N\ell t_{\text{AES}}/128$ (as we generates 3 random vectors).

Our *2-party* Shuffle protocol with Pre-processing protocols. By simple counting, in our pre-processing protocol for shuffle (see the full version for the details), the dominant running time consists of 3 executions of the Permute+Share protocol of [9]. When applying the parallelization technique of sequential Permute+Share protocol execution (see Section B.2), the offline communication time is $(6dk \log T + 3(d+1)\ell + 32)N/B$. Setting the same parameters as the original CGP, we have the offline communication equals to $2T_1 - \frac{(\ell \log N - 64)N}{2B}$. On the other hand, the local computation time is roughly twice that of the modified CGP protocol. The online communication complexity is the same as in the client-aided setting.

B.4 Our Implementation for Shuffle Protocol

We implemented the full shuffle protocols (with $\ell = 64$). For our implementations, we used C++ and ran the measurements on 22-core machines with 756GB RAM, Ubuntu 16.04 LTS, and GCC compiler version 9.2.1. The benchmark result is given in Table 4. This confirms our estimation time in Figure 3 (plus some extra time naturally occurred in implementations).

Table 6: The online communication complexity of subprotocols in [11]

Sub-protocol	# com. bits	# rounds
ModConv	L	1
GenMultiBitSort	$(2^L - L)\lambda$	$\lceil \log_2 L \rceil + 1$
OptApplyInv	$4\lambda + 3L$	3
OptCompose	2λ	2

Table 5: Communication required for each step of $\Pi_{\mathsf{RadixSort}}$ in the online phase

Step	Protocol for	# com. bits	# rounds
2	GenBitsPerm	$(2^L - 1)\lambda + L$	2
3	Shuffle	$\lambda + L$	1
	Reconst	λ	2
	GenBitsPerm	$(2^L - 1)\lambda + L$	
4	Shuffle	λ	1

B.5 Detailed Analysis for Our Sorting Protocol

Our Sorting protocol in *Client-aided* setting. We first measure the communication complexity of our protocol $\Pi_{\mathsf{RadixSort}}$. Table 5 shows the numbers of online communication bits (per party and record) and rounds required for each step. We note that since the first communication round of Reconst and $\Pi_{\mathsf{GenBitsPerm}}$ can be run in parallel, each execution of Step 3 takes three rounds in total. To sum up, the online communication complexity of $\Pi_{\mathsf{RadixSort}}$ is $\lceil \frac{\ell}{L} \rceil (2^L + 1)\lambda - \lambda + 2\ell - L$ bits and $3\lceil \frac{\ell}{L} \rceil$ rounds. In particular, the round complexity is improved from Chida *et al.* [11], where it was $(\lceil \log_2 L \rceil + 6)\lceil \frac{\ell}{L} \rceil - 4$.

Considering the communication time, the optimal choice of L depends on the setting. While the number of communication rounds decreases as the value of L increases, the communication bits are at a minimum when $L = 2$. When N is so small that we can ignore the data transfer time, we can set L to be larger. When N is large, it is good to set $L = 2$. In our estimation, we set $L = 2$ or $L = 3$.

We next consider the offline complexity of $\Pi_{\mathsf{RadixSort}}$. During the offline phase, the parties invoke single Radix-prep, GBP-prep for each GenBitsPerm, and Shuffle-prep for each Shuffle. Assuming that each permutation is represented as an element of $\mathbb{Z}_{2^\lambda}^N$, and ℓ is divisible by L, the total memory costs for randomness per record and party is $3\lceil \frac{\ell}{L} \rceil (2^L + 1)\lambda - 6\lambda + 3\ell - 2L$ bits. When applying the pseudorandom secret sharing technique, the offline communication complexity is $2\lceil \frac{\ell}{L} \rceil (2^L + 1)\lambda - 5\lambda + \ell - L$ bits per record. By simple counting, our local computation time can be estimated as $(5\lceil \frac{\ell}{L} \rceil - 7)t_{\mathsf{perm}}$. We plot our *total* running time estimation compared to those of the three-party protocol of [11] in Figure 7, using $\ell = 64$ and $\lambda = 32$.

Our *2-Party* Sorting protocol with Pre-processing protocols. Unfortunately, the offline phase of our sorting protocol is, when instantiated in the (strict) two-party setting (see the full version for the details), not as practical as its online phase. Its offline phase needs to generate a bunch of correlated randomnesses used by protocols in the online phase realizing Shuffle and for GenBitsPerm. The preprocessing for Π_{Shuffle} and that for a protocol realizing GenBitsPerm can be run parallelly. These protocols for preprocessing result in several (constant-number) parallel executions of $O(\ell/L)$ instances of the Permute+Share protocol by Chase et al. [9], which in turn involves public-key type operations. Thus, although the round complexity of the resulting offline protocol can be constant by appropriately using parallelism, its efficiency is no longer competitive to the offline phase of Chida et al.'s three-party sorting protocol which uses no public-key type operations.

B.6 Recapped Analysis for the Chida *et al.* Protocols

This subsection recapitulates the analysis of running time for the Chida *et al.* shuffle and sort protocols [11].

The CHI+ *3-party* Shuffle protocol. The OptShuffle protocol in [11] utilizes $\mathcal{F}_{\mathsf{rand}}$ as a preprocessing functionality. Applying the pseudorandom secret sharing technique can omit the offline communication for $\mathcal{F}_{\mathsf{rand}}$. In preprocessing phase, each party needs to obtain two random permutations and two random vectors by calling $\mathcal{F}_{\mathsf{rand}}$. Hence, the offline computation time is $2t_{\mathsf{perm}} + 2N\ell t_{\mathsf{AES}}/128$. In the online phase, this protocol takes two communication rounds, and the communication time for each communication round is $\frac{\ell N}{B}$. The plot of the online running time of the OptShuffle protocol is the same as the ModifiedCGP protocol in Figure 4. Figure 8 provides a plot of the total running time of the OptShuffle protocol and a comparison to our protocol. In this comparison, our shuffle protocol is around 0.91 times faster on average.

The CHI+ *3-party* Sort protocol. We can regard the whole protocol called OptGenPerm in [11] as the online phase except for the calls to two functionalities called $\mathcal{F}_{\mathsf{rand}}$ and $\mathcal{F}_{\mathsf{doublerand}}$ in [11]. Table 6 shows the online communication complexity of subprotocols in [11]. By inspection, one can derive that the online communication complexity is $(\lceil \log_2 L \rceil + 6)\lceil \frac{\ell}{L} \rceil - 4$ rounds and $\lceil \frac{\ell}{L} \rceil (2^L - L + 6)\lambda - 6\lambda + 3\ell - 2L$ bits per record. We note that data transmission in protocols of [11] is not symmetric, hence the communication bits here are counted by choosing the maximum communication bits on all channels for each round. We plot the case $L = 3$ in Figure 6, which is known to minimize the communication bits for [11].

In the offline phase, it is required to generate randomness of $\lceil \frac{\ell}{L} \rceil (2^{L+1} + L + 6)\lambda - 8\lambda + 5\ell - 4L$ bits per record and party, when counted as in Section B.5. Using the pseudorandom secret sharing technique, each party only needs to send $2\ell\lambda$ bits per record in offline (when using the protocol for $\mathcal{F}_{\mathsf{doublerand}}$ in [24]). Note that the local computation time is dominant by the cost of generating random permutations, and by simple counting, we have that it is $2(\lceil \frac{\ell}{L} \rceil - 1)t_{\mathsf{perm}}$ in [11]; while ours is $(5\lceil \frac{\ell}{L} \rceil - 7)t_{\mathsf{perm}}$.

C MORE INSTANCES OF GROUP ACTIONS

C.1 More Examples of Linear Group Actions

Example 5. Polynomial Translation Action. Let $a \in R$. We define $M_a^{(n)}$ to be the $(n+1) \times (n+1)$ matrix whose entry is given

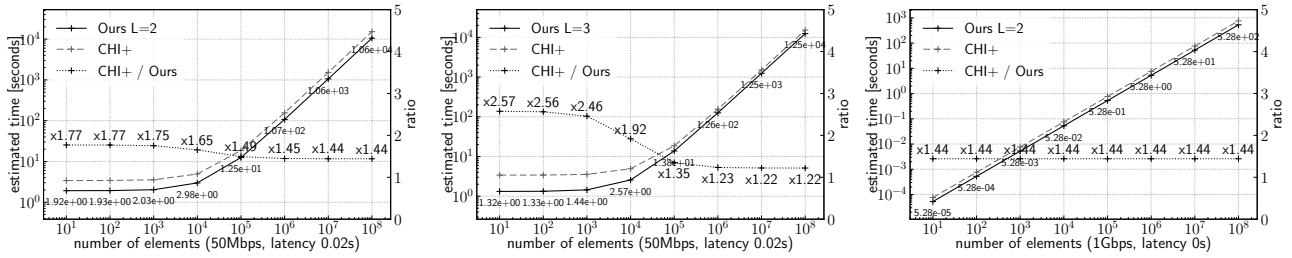

Figure 6: *Online* running times for our *oblivious sort* protocols, compared to Chida *et al.* (CHI+ [11]), with improvement ratios

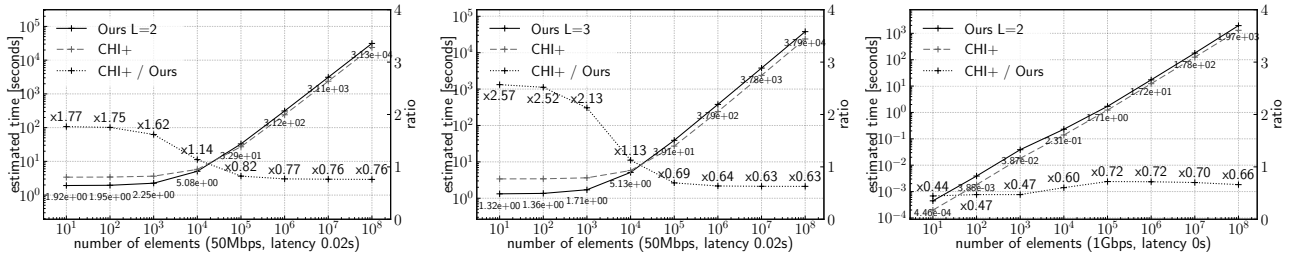

Figure 7: *Total* running times for our *oblivious sort* protocols, compared to Chida *et al.* (CHI+ [11]), with improvement ratios

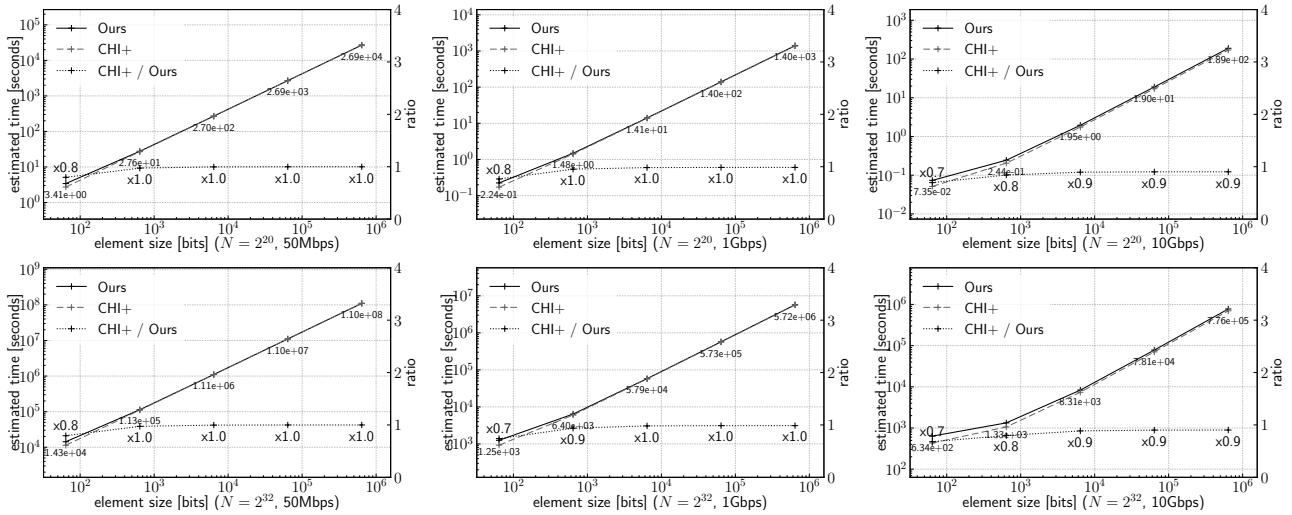

Figure 8: *Total* running times for our *oblivious shuffle* protocols, compared to Chida *et al.* (CHI+ [11]), with improvement ratios

by

$$m_{i,j} = \begin{cases} 0 & \text{if } i > j \\ \binom{j}{i} a^{j-i} & \text{otherwise} \end{cases}$$

where $\binom{n}{k}$ denotes the binomial coefficient. We can easily check that $M_0^{(n)} = E$ and $M_a^{(n)} M_b^{(n)} = M_b^{(n)} M_a^{(n)} = M_{a+b}^{(n)}$ for all $a, b \in R$. Hence, the set of such matrices is commutative and closed under multiplication. Define the function $\psi : R^{n+1} \times R \to R^{n+1}$ as follows:

$$\boldsymbol{v} \cdot a := M_a^{(n)} \boldsymbol{v}$$

Clearly, the function ψ gives a linear group action of R on R^{n+1}.

We note that $M_a^{(n)}$ corresponds to the linear operator $(T_a f)(x) = f(x+a)$ on the vector space of polynomials of degree n. Let $f(x) = \sum_{i=0}^{n} c_i x^i$. Then, the coefficients in the expansion of the translated polynomial $f(x+a)$ are calculated by the following matrix multiplication:

$$\begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_n \end{pmatrix} = M_a^{(n)} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix}$$

where d_i is the i-th coefficient, *i.e.*, $f(x+a) = \sum_{i=0}^{n} d_i x^i$.

Example 6. Linear Group Action by Direct Product Group. Let (G_1, ψ_1) and (G_2, ψ_2) be linear group actions on R-modules M_1

and M_2, respectively. We define the function $(\psi_1 \otimes \psi_2)$ as follows:

$$\psi_1 \otimes \psi_2 : (M_1 \otimes M_2) \times (G_1 \times G_2) \to (M_1 \otimes M_2)$$
$$(\sum (x \otimes y)) \cdot (g_1, g_2) := \sum ((x \cdot g_1) \otimes (y \cdot g_2))$$

where $\otimes$ denotes a tensor product of R-modules. We can easily check that the function $(\psi_1 \otimes \psi_2)$ is well-defined as follows:

$$\begin{aligned}((x_1 + x_2) \otimes y) \cdot (g_1, g_2) &= ((x_1 + x_2) \cdot g_1) \otimes (y \cdot g_2) \\ &= ((x_1 \cdot g_1) + (x_2 \cdot g_1)) \otimes (y \cdot g_2) \\ &= (x_1 \cdot g_1) \otimes (y \cdot g_2) + (x_2 \cdot g_1) \otimes (y \cdot g_2) \\ &= (x_1 \otimes y + x_2 \otimes y) \cdot (g_1, g_2)\end{aligned}$$

Similarly, we have $(x \otimes (y_1 + y_2)) \cdot (g_1, g_2) = (x \otimes y_1 + x \otimes y_2) \cdot (g_1, g_2)$ and $(rx \otimes y) \cdot (g_1, g_2) = r(x \otimes y) \cdot (g_1, g_2)$. Clearly, the function $(\psi_1 \otimes \psi_2)$ forms a linear group action of the direct product $(G_1 \times G_2)$ on $(M_1 \otimes M_2)$.

C.2 More Instances of Our Main Protocol

Oblivious Polynomial Translation. Another typical instance of LGA is oblivious polynomial translation. We write Trans and Π_{Trans} for the functionality and the corresponding protocol, respectively, in the case the linear group action is the polynomial translation action (see Example 5 in Section C.1). Since G is commutative, a double-sharing of r is represented as $\langle r \rangle = (r_1, r_2)$ for some r_1, r_2. Trans takes a secret-shared polynomial $[f]$ represented by a sequence of secret-shared coefficients as input, and outputs a sharing $[g]$ of the polynomial g whose variable is translated by r, *i.e.,* $g(x) = f(x + r)$. In particular, as a special case, we can obtain a sharing $[rv]$ for given $\langle r \rangle$ and $[v]$.

D MORE APPLICATIONS

This section describes more applications, as deferred from Section 1 (Further Results). We briefly summarize the results here as follows: we obtain online-one-round protocols for the following computations.

- Oblivious Multiplexer. This is defined as a boolean version of Oblivious Selection (*cf.* Section 5.1), where here the selector value is bit-decomposed and is in the boolean-shared form (see below). To multiplex a 2^m-element vector, our protocol runs in one round and requires online communication of $2^m - 1$ elements and m bits (this is comparable to our protocol for oblivious selection in Section 5.1). To the best of our knowledge, this is the first explicit secret-sharing-based online-one-round protocol for oblivious multiplexer. Note that one can obtain oblivious multiplexer using one boolean-to-arithmetic (B2A) conversion and one oblivious selection (sequentially), but this would require at least 2 rounds. As a particularly useful special case, when $m = 1$, we achieve a secure protocol for "if-then" conditional computation, which is very commonly used in many applications. Our secure "if-then" protocol is very efficient: the online communication requires only 1 element and 1 bit. See more related works below.
- Oblivious Polynomial Evaluation. We consider the "secret-shared" variant of oblivious polynomial evaluation, where the input value, the polynomial, and the output are all in the secret-shared form (see below), in contrast to *e.g.,* [19, 35], which considers the "plain" variant. Note that the secret-shared variant is harder to achieve, while also is particularly useful as it can be used as sub-protocols in more complex computations. To the best of our knowledge, ours is the first explicit secret-sharing-based online-one-round protocol (for this variant). As for its efficiency, to securely compute any secret-shared polynomial of degree n, each party simply sends $n + 1$ elements to each other, in one round.
- Secure (Multi-input) Multiplication, which is a particular case of a direct product of Oblivious Polynomial Translation. This protocol achieves the same communication cost as the (multi-input) generalized Beaver triple based online-one-round method recently proposed in [37]. Note that, to the best of our knowledge, before the work of [37], there exists some secret-sharing-based online-constant-round protocols for secure multi-input multiplication albeit only with the constants being larger than one*e.g.,* [8, 13, 33, 36].[12] In particular, this demonstrates the generality of our framework (in capturing previous efficient protocols as a special case).

Related Works. Multiplexers for general m-element vectors can be viewed as similar to protocols called "private lookup" or "private indexing", which obliviously select a single element from an array of data (like databases) while keeping the data and the index secret. Oblivious RAM or private information retrieval (PIR) are known as versatile techniques to realize private lookup. The paper of [25] proposed private lookup for secret-shared data with $O(1)$ communication complexity, though its round complexity is $O(n)$. Similar idea can be seen in MPC protocols for deterministic finite automata (DFA) [45] to hide access pattern. Private indexing is also implemented as multiplexers of m-element vector in several MPC compilers, such as CBMC-GC [20] and PICCO [48]. However, to the best of our knowledge, there is no known constant-round schemes based on such techniques for secret-shared data.

D.1 Oblivious Multiplexer

By using the XORShuffle protocol, we can construct a boolean version of our oblivious selection protocol. The functionality Mux is defined as: $\mathsf{Share}(\boldsymbol{v}_n) \leftarrow \mathsf{Mux}([\boldsymbol{v}], [n_m]^B, \ldots, [n_1]^B)$, where $n = \sum_{i=1}^{m} 2^{i-1} n_i + 1$.

We describe a protocol for computing oblivious multiplexer circuits. Here, '$\cdot$' denotes the XOR permutation action.

Protocol 11. Π_{Mux}

Input: $([\boldsymbol{v}]_i, [n_m]_i^B, \ldots, [n_1]_i^B)$ from P_i, where $\boldsymbol{v} \in D^{(2^m)}$.
Output: $[\boldsymbol{v}_n]_i$ for P_i, where $[\boldsymbol{v}_n] \leftarrow \mathsf{Share}(\boldsymbol{v}_n)$ and $n = \sum_{i=1}^{m} 2^{i-1} n_i + 1$.
Subfunctionality: XORShuffle.

(1) Each party P_i samples $r_{i,j} \xleftarrow{\$} \{0, 1\}$ for $j \in \{1, \ldots, m\}$ and sets $\langle r \rangle_i := (r_{i,m}, \ldots, r_{i,1})$.
(2) The parties invoke $[\boldsymbol{w}] \leftarrow \mathsf{XORShuffle}([\boldsymbol{v}]; \langle r \rangle)$.
(3) The parties compute $p_j \leftarrow \mathsf{Reconst}([n_j]^B \oplus [r_j]^B)$ for $j \in \{1, \ldots, m\}$.
(4) Each party locally computes $[z] := [\boldsymbol{w}_p]$ for $p = \sum_{i=1}^{m} 2^{i-1} p_i + 1$ and outputs it.

The correctness is shown as follows:

$$\boldsymbol{w}_p = (\boldsymbol{w} \cdot g)_1 = (\boldsymbol{v} \cdot (r \circ g))_1 = (\boldsymbol{v} \cdot (n_m, \ldots, n_1))_1 = \boldsymbol{v}_n,$$

[12] Moreover, these latter systems can be used over finite fields, while the work of [37] and ours can also be used in any rings (which is more general).

where $g := (p_m, \ldots, p_1)$. The protocol takes only one round in the online phase since Steps 2-3 can be run in parallel.

Using the optimized XORShuffle protocol, each party is required to send m bits and $2^m - 1$ elements. In particular, our Mux protocol of the case $m = 1$ corresponds to the "if-then" protocol, in which each party needs to send one bit and one element in the online phase.

The security of Π_{Mux} is guaranteed as follows. We omit its proof since it is similar to the proof of Theorem 4.1.

Theorem D.1. *The protocol Π_{Mux} for computing Mux is prefectly semi-honest secure in the* XORShuffle*-hybrid model.*

D.2 Oblivious Polynomial Evaluation

Our proposed framework for linear group action also includes a secret-sharing variant of oblivious polynomial evaluation. The functionality Polyval is defined as: $\mathsf{Share}(v) \leftarrow \mathsf{Polyval}([x], [c_0], \ldots, [c_n])$, where $v = \sum_{j=0}^{n} c_j x^j$.

We describe a protocol for oblivious polynomial evaluation. Here, '$\cdot$' denotes the polynomial translation action of degree n.

Protocol 12. Π_{Polyval}

Input: $[c_j]_i, [x]_i$ from P_i, where $c_j \in R$ for $j \in \{0, \ldots, n\}$, $x \in R$.
Output: $[z]_i$ for P_i, where $z = \sum_{j=0}^{n} c_j x^j$ and $[z] \leftarrow \mathsf{Share}(z)$.
Subfunctionality: Trans.

(1) Each party P_i samples $r_i \xleftarrow{\$} R$ and sets $\langle r \rangle_i := r_i$.
(2) The parties invoke $[\boldsymbol{v}] \leftarrow \mathsf{LGA}(([c_0], [c_1], \ldots, [c_n])^\top; \langle r \rangle)$.
(3) The parties compute $p := \mathsf{Reconst}([x] - [r])$.
(4) Each party P_i locally computes $[\boldsymbol{w}]_i := [\boldsymbol{v}]_i \cdot p$ and outputs $[z]_i := [\boldsymbol{w}_0]_i$.

Correctness is shown as follows:

$$\boldsymbol{w} = M_p^{(n)} \boldsymbol{v} = M_{x-r}^{(n)} M_r^{(n)} \boldsymbol{c} = M_x^{(n)} \boldsymbol{c}$$

where $\boldsymbol{c} = (c_0, \ldots, c_n)^\top$. By the definition of $M_x^{(n)}$, we have $\boldsymbol{w}_0 = \sum_{j=0}^{n} c_j x^j$.

The protocol takes one round in the online phase since Steps 2-3 can be run in parallel. By applying the communication reduction technique (see the full version for details), each party needs to send $n + 1$ elements, and needs to store $2n + 1$ elements for correlated randomness.

We note that Polyval includes secure multiplication as a special case of $n = 1$: Let $c_0 = 0$ and $[c_1] = [y]$, then the output of Polyval is $[xy]$. In this case, our protocol Π_{Polyval} requires the same numbers of communication and storage bits as the Beaver triple-based multiplication protocol.

The security of Π_{Polyval} is guaranteed as follows. We omit its proof since it is similar to the proof of Theorem 4.1.

Theorem D.2. *The protocol $\Pi_{Polyval}$ for computing Polyval is prefectly semi-honest secure in the Trans-hybrid model.*

D.3 Secure (Multi-input) Multiplication

Considering the linear group action by a direct product group instead of each component group individually, we can obtain round-efficient protocols. As an example, we provide a one-round protocol for secure multiplication with multiple arguments. The functionality Mult is defined as: $\mathsf{Share}(v) \leftarrow \mathsf{Mult}([n_1], \ldots, [n_m])$, where $v = \prod_{j=1}^{m} n_j$.

Let ψ be the polynomial translation of degree 1. Consider the linear group action derived from $\psi^{\otimes m} : R^{2^m} \times R^m \to R^{2^m}$ (see Example 6 in Section C.1). We note that Beaver triple can be represented in terms of the linear group action. When $m = 2$, we have

$$M_a^{(1)} \otimes M_b^{(1)} = \begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & b \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & a & b & ab \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & a \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Therefore, the output of $\mathsf{LGAp}(\boldsymbol{e}^{(4)}; \perp)$ is $([r_1], [r_2], ([r_1 r_2], [r_2], [r_1], [1])^\top)$ for some $r_1, r_2 \in R$. When we omit the sharings of duplicate values and constants, the output consists of $[r_1]$, $[r_2]$ and $[r_1 r_2]$, which correspond to a Beaver triple.

We describe a protocol for secure multiplication of multiple arithmetic sharings. Here, '$\cdot$' denotes $\psi^{\otimes m}$ described above.

Protocol 13. Π_{Mult}

Input: $[n_1]_i, \ldots, [n_m]_i$ from P_i, where $n_j \in R$ for $j \in \{1, \ldots, m\}$.
Output: $[v]_i$ for P_i, where $v = \prod_{j=1}^{m} n_j$.
Subfunctionality: LGA_p.

(1) The parties invoke $([r_1], \ldots, [r_m], \mathsf{bt}) \leftarrow \mathsf{LGA\_p}(\boldsymbol{e}^{(2^m)})$.
(2) The parties compute $p_i := \mathsf{Reconst}([n_i] - [r_i])$ for $i \in \{1, \ldots, m\}$.
(3) Each party locally computes $[\boldsymbol{w}]_i := \mathsf{bt}_i \cdot (p_1, \ldots, p_m)$.
(4) Each party locally computes $[v]_i := [\boldsymbol{w}_1]$.

Note that the protocol has one round in the online phase since Steps 1 can be offline. The number of elements each party sends in online phase is m. Correctness is shown as follows:

$$\begin{aligned} \boldsymbol{w} &= (\otimes_{j=1}^{m} M_{p_j}) \mathsf{bt} = (\otimes_{j=1}^{m} M_{n_j - r_j})(\otimes_{j=1}^{m} M_{r_j}) \boldsymbol{e}^{(2^m)} \\ &= (\otimes_{j=1}^{m} M_{n_j}) \boldsymbol{e}^{(2^m)}, \end{aligned}$$

By definition, we have $\boldsymbol{w}_1 = \prod_{j=1}^{m} n_j$.

The security of Π_{Mult} is guaranteed as follows. We omit its proof since it is similar to the proof of Theorem 4.1.

Theorem D.3. *The protocol Π_{Mult} for computing Mult is prefectly semi-honest secure in the LGA-hybrid model.*