Multi-party Computation with Small Shuffle Complexity Using Regular Polygon Cards

Kazumasa Shinagawa[1,2](✉), Takaaki Mizuki[3], Jacob C.N. Schuldt[2], Koji Nuida[2], Naoki Kanayama[1], Takashi Nishide[1], Goichiro Hanaoka[2], and Eiji Okamoto[1]

[1] University of Tsukuba, Tsukuba, Japan
shinagawa@cipher.risk.tsukuba.ac.jp
[2] National Institute of Advanced Industrial Science and Technology, Tsukuba, Japan
[3] Tohoku University, Sendai, Japan

Abstract. It is well-known that a protocol for any function can be constructed using only cards and various shuffling techniques (this is referred to as a *card-based protocol*). In this paper, we propose a new type of cards called regular polygon cards. These cards enable a new encoding for multi-valued inputs while the previous works can only handle binary inputs. We furthermore propose a new technique for constructing a card-based protocol for any n-ary function with small shuffle complexity. This is the first general construction in which the shuffle complexity is independent of the complexity (size/depth) of the desired functionality, although being directly proportional to the number of inputs. The construction furthermore supports a wide range of cards and encodings, including previously proposed types of cards. Our techniques provide a method for reducing the number of shuffles in card-based protocols.

Keywords: Multi-party computation · Card-based protocol · Polygon cards · Shuffle complexity

1 Introduction

1.1 Background

Since the seminal work of den Boer [2], many card-based protocols have been proposed, which can securely compute a function by applying shuffles to sequences of cards [1–13]. Compared to computer-based protocols, a card-based protocol can be performed without the use of computers and electricity. Thus, this type of protocol is suitable when computers are not available or the parties do not trust the security of computers-based protocols (although card-based protocols require a different set of trust assumptions). Moreover, it is easy to understand the correctness and the security of card-based protocols since they do not rely on complicated reductions to mathematical problems which may be hard to verify and understand for non-experts.

The *Five-Card Trick* [2] is the first card-based protocol, in which two parties can securely compute the AND function of their secret inputs, using five cards that have two types of front sides ($\boxed{\clubsuit}$, $\boxed{\heartsuit}$) and identical backs ($\boxed{\times}$). In the subsequent works [1,3–13], many card-based protocols are proposed which focus on feasibility results and reducing the number of cards required in the protocols. In 2009, Mizuki and Sone [8] proposed composable AND, XOR, and COPY protocols using six, four, and six cards, respectively. (It is possible to compute any functions by composing these protocols.) These results [8] are the most efficient construction for the elementary boolean functions with respect to a commonly used encoding scheme in which each input bit is encoded using two cards (a *two-cards-per-bit encoding scheme*). For any n-ary boolean function, Nishida et al. [10] showed that it is possible to construct a $(2n+6)$-card protocol using a two-cards-per-bit encoding scheme. In 2014, Mizuki and Shizuya [7] showed that under a *one-card-per-bit encoding scheme*[1] it is possible to construct composable AND, XOR, and COPY protocols using three, two, and three cards with rotationally symmetric backs. (A protocol for any n-ary function using only $n + 3$ cards can easily be obtained by combining the result from [10] with the encoding scheme in [7].)

While previous results show that it is feasible to construct a protocol for an arbitrary function using a small number of cards, it is unknown how to construct a protocol with small shuffle complexity. Since shuffles are the most costly operations, a large number of shuffles immediately imply a large computational overhead. Let f be a function and let $|f|$ be the smallest number of gates (AND/XOR/COPY) in circuits implementing f. In this case, we can obtain a protocol for f using the previous AND/XOR/COPY protocols to evaluate the circuit for f, which yields a shuffle complexity of exactly $|f|$. We stress that even if a protocol has an asymptotic small number of shuffles (e.g. polynomial in $|f|$), it is not always considered to be efficient. For card-based protocols, it is desirable that the shuffle complexity is as low as possible (with small coefficients), and ideally, that it is independent of the complexity of the desired functionality.

1.2 Our Contribution

Our main contributions are as follows: (1) We first propose a new type of cards, regular polygon cards, that can deal with multi-valued inputs directly. As a result, we can construct a protocol for an arbitrary linear function with small shuffle complexity. (2) We define a new notion, which we call oblivious conversion. This enables the construction of a protocol for any functions with small shuffle complexity. (3) We show that the regular polygon cards enable the construction of an efficient voting protocol for multiple candidates. Our protocols

[1] We stress that the two-card-per-bit encoding schemes are important since the one-card-per-bit encoding scheme [7] needs unnatural shuffle for computing the AND function. Thus, it is still meaningful to improve protocols under the two-cards-per-bit encoding schemes.

Table 1. Comparison between our protocols and previous protocols

	Type of cards	Shuffle Complexity	Number of cards
∘ Addition and Subtraction over $\mathbb{Z}/m\mathbb{Z}$			
[3,8] based	standard	$O(\log m)$	$O(\log m)$
Ours	m-sided	1	2
∘ Multiplication by $a \in \mathbb{Z}/m\mathbb{Z}$			
[3,8] based	standard	$O(\log a \cdot \log m)$	$O(\log a \cdot \log m)$
Ours	m-sided	$\lceil \log_2 a \rceil + 1$	$\lceil \log_2 a \rceil + 2$
∘ Protocol for an arbitrary $f : (\mathbb{Z}/m\mathbb{Z})^n \to \mathbb{Z}/m\mathbb{Z}$			
[10] based	standard	$O(m^n \cdot \log m)$	$2((n+1)\lceil \log_2 m \rceil + 2)$
Ours	m-sided	n	$mn + m^n$
∘ Protocol for an arbitrary $f : (\mathbb{Z}/2\mathbb{Z})^n \to \mathbb{Z}/2\mathbb{Z}$			
[10]	standard	$O(2^n)$	$2(n+3)$
Ours	standard	n	$2(n+2^n)$

Table 2. Comparison between our voting protocol and previous voting protocol

	# of voters	# of candi.	Input timing	# of shuffles	# of cards
[3] (standard)	any n	2	restricted	$O(n \log n)$	$2\lceil \log_2 n \rceil + 6$
Ours (Sect. 4)	n $(n < m)$	any ℓ	no restriction	n	$\ell^n + \ell n$
Ours (Sect. 5)	n $(n < m)$	any ℓ	no restriction	$n+1$	$(n+2)\ell$

using regular m-sided polygon cards have smaller shuffle complexity than protocols using the previously proposed cards $\boxed{\clubsuit}$, $\boxed{\heartsuit}$ (the cards are referred to as *standard* in Table 1).

Regular Polygon Cards. The regular m-sided polygon cards (Fig. 1) have $(360/m)^\circ$ rotational symmetry, and have a value corresponding to an element of $\mathbb{Z}/m\mathbb{Z}$. The card with rotationally symmetric backs proposed by Mizuki and Shizuya [7] (in the context of one-card-per-bit encoding schemes) can be regarded as a regular 2-sided polygon card. Our work introduces the first card-based protocols for multi-valued inputs while all previous works [1–13] only consider binary inputs. Using the m-sided polygon cards, it is possible to construct addition, subtraction, and copy protocols over $\mathbb{Z}/m\mathbb{Z}$ using only a single shuffle while protocols based on [8,10] use $O(\log m)$ shuffles (see Table 1, Addition and Subtraction). We also construct a protocol for multiplication by a constant $a \in \mathbb{Z}/m\mathbb{Z}$ using only $\lceil \log_2 a \rceil + 1$ shuffles while protocols based on [8,10] use $O(\log a \cdot \log m)$ shuffles (see Table 1, Multiplication). Composing our protocols, we can securely compute an arbitrary linear function while maintaining a small shuffle complexity.

Oblivious Conversion. We define a new notion, oblivious conversion, which is a generalization of oblivious transfer. This is a protocol that takes as inputs an encoding of $x \in \mathbb{Z}/m_0\mathbb{Z}$ and a function $f : \mathbb{Z}/m_0\mathbb{Z} \to \mathbb{Z}/m_1\mathbb{Z}$, and outputs an encoding of $f(x) \in \mathbb{Z}/m_1\mathbb{Z}$. By applying an oblivious conversion n times, it is possible to construct protocols for arbitrary n-ary functions (see Table 1, Protocol for an arbitrary f). This approach can be applied to various cards (and encodings) including the regular polygon cards and the previously proposed cards ($\boxed{\clubsuit}$,$\boxed{\heartsuit}$). Here, a protocol for $f : \bigotimes_{i=0}^{n-1} \mathbb{Z}/m_i\mathbb{Z} \rightarrow \mathbb{Z}/m_n\mathbb{Z}$ (i.e., the domain of f is n tuples in which each element belongs to $\mathbb{Z}/m_i\mathbb{Z}$) is regarded as an MPC protocol for f with n parties $P_0, P_1, \cdots, P_{n-1}$ where P_i holds the secret input $x_i \in \mathbb{Z}/m_i\mathbb{Z}$. Thus, an MPC protocol for any n-ary function can be obtained by applying oblivious conversion by n times. The shuffle complexity of the obtained protocol is small, and it does not depend on the complexity (size/depth) of the function to be evaluated.

Voting Protocol. While oblivious conversion allows the generic construction of protocols with a small shuffle complexity, it might still be possible to construct even more efficient protocols for specific functionalities. We specifically construct an efficient voting protocol for multiple candidates. For n voters and ℓ candidates, using the regular m-sided polygon cards ($m > n$), our protocol uses $n+1$ shuffles and $(n+2)\ell$ cards, while a protocol based on oblivious conversion uses n shuffles and $\ell^n + \ell n$ cards.

1.3 Related Works

In 1993, Crépeau and Kilian [1] achieved protocols implementing any functionality by constructing composable elementary protocols (COPY/XOR/AND). In 2009, Mizuki and Sone [8] constructed composable elementary protocols using fewer cards, by applying a new shuffle called a *random bisection cut*. To evaluate a function that has $|f|$ gates (COPY/XOR/AND), the shuffle complexity of the obtained protocol is exactly $|f|$. On the other hand, our construction (Sect. 4) requires only n shuffles where n is the number of inputs to the function.

Mizuki, Asiedu, and Sone [3] constructed a voting protocol with n voters and 2 candidates, using $2\lceil \log_2 n \rceil + 6$ cards. However, this protocol restricts the timing of the voter inputs in order to reduce the number of required cards; an unrestricted protocol requires $O(n)$ cards to encode the voter inputs. Our protocol is unrestricted, requires n cards, and makes use of $n+1$ shuffles. In contrast, the protocol from [3] requires $O(n \log n)$ shuffles (see Table 2).

2 Our New Cards and Model of Protocols

In this section, we propose *regular polygon cards*, and define protocols and the security based on regular polygon cards as well as the standard notion of security.

2.1 Regular Polygon Cards

We first propose new cards called *regular polygon cards*, which are conceptually different from previous cards ($\clubsuit$, $\heartsuit$). A regular m-sided polygon card encodes an element of $\mathbb{Z}/m\mathbb{Z}$, while previous works use two cards to encode an element of $\mathbb{Z}/2\mathbb{Z}$. From now on, we use $\mathbb{Z}_m$ to denote $\mathbb{Z}/m\mathbb{Z}$.

Definition 1 (Regular Polygon Card). *A card is called a* regular m*-sided* polygon card *if the front side of the card has no rotational symmetry and the back side of the card has* $(360/m)^\circ$ *rotational symmetry.*

Note that m, in a regular m-sided polygon card, does not refer to the shape of the cards, but the symmetry of the back side of the card. Indeed, for all $m = m_0 m_1$, a regular m-sided polygon card is both a regular m_0-sided polygon card and a regular m_1-sided polygon card. In the case of $m = 2$, a regular 2-sided polygon card refer to a card with a 180° rotationally symmetric pattern [7].

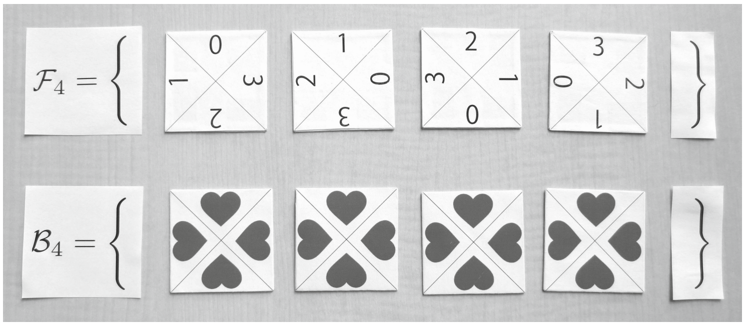

Fig. 1. An example of regular 4-sided polygon cards

Set of Cards. The cards (Fig. 1) are an example of regular m-sided polygon cards ($m = 4$). Letting the front side of a card correspond to a value of $\mathbb{Z}_m$ decided by the card's rotation, a set of front side cards $\mathcal{F}_m$ is obtained as $\mathcal{F}_m = \{0, 1, \cdots, m-1\}$. (See the upper four cards in Fig. 1; the value of a card corresponds to the number shown in the top of the card.) Using $[\![x]\!]$ to denote a card put face-down[2] whose value is x, a set of back side cards $\mathcal{B}_m$ is obtained as $\mathcal{B}_m = \{[\![0]\!], [\![1]\!], \cdots, [\![m-1]\!]\}$ (see the lower four cards in Fig. 1). A set of cards $\mathcal{C}_m$ is defined by $\mathcal{C}_m = \mathcal{F}_m \cup \mathcal{B}_m$. Let rot_m and flip_m be a rotation function and a flip function, s.t., $\mathsf{rot}_m, \mathsf{flip}_m : \mathcal{C}_m \rightarrow \mathcal{C}_m$. The rotation function rot_m takes $x \in \mathcal{F}_m$ or $[\![x]\!] \in \mathcal{B}_m$ as input, and outputs $x - 1 \in \mathcal{F}_m$ or $[\![x+1]\!] \in \mathcal{B}_m$, respectively. (Note that both of $x \rightarrow x - 1$ and $[\![x]\!] \rightarrow [\![x+1]\!]$ correspond to the same rotation operation.) The flip function flip_m takes $x \in \mathcal{F}_m$ or $[\![x]\!] \in \mathcal{B}_m$ as input, and outputs $[\![x]\!] \in \mathcal{B}_m$ or $x \in \mathcal{F}_m$, respectively. From now on, we often omit m and denote $\mathcal{C}_m, \mathcal{F}_m, \mathcal{B}_m, \mathsf{rot}_m, \mathsf{flip}_m$ as $\mathcal{C}, \mathcal{F}, \mathcal{B}, \mathsf{rot}, \mathsf{flip}$. Using rot and flip, any card $[\![x]\!] \in \mathcal{B}$ or $x \in \mathcal{F}$ can be expressed by operations on $[\![0]\!]$, i.e., $[\![x]\!] = \mathsf{rot}^x([\![0]\!])$ and $x = \mathsf{flip}(\mathsf{rot}^x([\![0]\!]))$. We define a *face function* face which expresses the face of cards. For $[\![x]\!] \in \mathcal{B}$, $\mathsf{face}([\![x]\!]) = \times$, and for $x \in \mathcal{F}$, $\mathsf{face}(x) =$ "x", where $\times$ and "x" are symbols. Thus, the face function is a function that takes an element of $\mathcal{C}$ as input, and outputs an element of a set of symbols $\{$"0", "1", $\cdots$, "$m-1$", $\times\}$.

[2] We assume that the direction of flipping is predetermined.

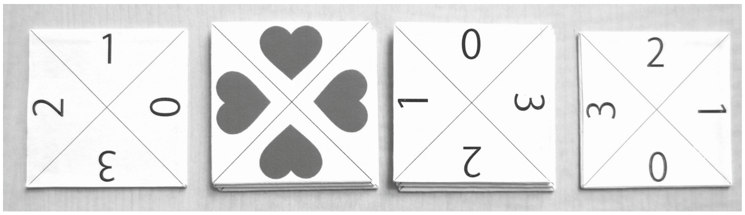

Fig. 2. An example of a sequence whose face is (“1”, $\times^3$, “0”^{3}, “2”)

Stack/Sequence. A *stack of cards* is defined by an ordered collection of cards. For t cards $c_0, c_1, \cdots, c_{t-1} \in \mathcal{C}$, a stack d is denoted by $d = c_0 \circ c_1 \circ \cdots \circ c_{t-1} \in \mathcal{C}^t$ (the top is c_0 and c_{t-1} is on the table), where $\mathcal{C}^t$ is the set of all stacks of t cards. Note that a card $c \in \mathcal{C}$ is a special case of a stack. We use $\mathcal{D}$ to denote a set of all stacks, i.e., $\mathcal{D} = \bigcup_{i=1}^{\infty} \mathcal{C}^i$. A *sequence (of stacks)* is defined by a vector of multiple stacks (see an example of a sequence in Fig. 2). For k stacks $d_0, d_1, \cdots, d_{k-1} \in \mathcal{D}$, a sequence $\boldsymbol{d}$ is denoted by $\boldsymbol{d} = (d_0, d_1, \cdots, d_{k-1}) \in \mathcal{D}^k$. The difference between a stack and a sequence is that a stack is a single object while a sequence consists of multiple objects. For a stack $d = c_0 \circ c_1 \circ \cdots \circ c_{t-1}$, the *face function* is defined by $\mathsf{face}(d) = (\mathsf{face}(c_0), t)$. This means that a face of stacks have two pieces of information; the face of the top card and the number of cards in the stack[3]. From now on, we use $(\mathsf{face}(c))^t$ to denote $(\mathsf{face}(c), t)$. For example, for an encoding $[\![x]\!] \in \mathcal{B}$ and a card $c \in \mathcal{C}$, $\mathsf{face}([\![x]\!] \circ c) = (\mathsf{face}([\![x]\!]))^2 = \times^2$. For a sequence $\boldsymbol{d} = (d_0, d_1, \cdots, d_{k-1}) \in \mathcal{D}^k$, the *face function* is defined by $\mathsf{face}(\boldsymbol{d}) = (\mathsf{face}(d_0), \mathsf{face}(d_1), \cdots, \mathsf{face}(d_{k-1}))$ (see Fig. 2).

Encoding. For a finite set X, an *encoding* is defined as an injective function E which maps $x \in X$ to a tuple of back side cards $\mathcal{B}^k$, where k is the length of the encoding. This is also called a *commitment* in previous card-based protocols. For an encoding E over $\mathbb{Z}_m$, we often omit “mod m”, e.g., we use $\mathsf{E}(x-1)$ to denote $\mathsf{E}(x - 1 \bmod m)$. The most natural encoding is the one that maps $x \in \mathbb{Z}_m$ to $\mathsf{E}(x) = [\![x]\!]$. Unless otherwise noted, this is the encoding we use. (In Sects. 4 and 5, we use other encodings to achieve a small shuffle complexity.)

2.2 Operations

We now define operations for sequences; permutation, rotation, flip, shuffle, composition/decomposition, and insert/delete.

Permutation. Let S_k be the symmetric group of degree k. For a permutation of k objects $\sigma \in S_k$, we define a *permutation operation* σ that takes as input a sequence $\boldsymbol{d} = (d_0, d_1, \ldots, d_{k-1}) \in \mathcal{D}^k$ and outputs $\sigma(\boldsymbol{d})$. We define a useful permutation $\mathsf{cyc}_k \in S_k$ that takes $(d_0, d_1, \ldots, d_{k-1})$, and outputs $(d_1, \ldots, d_{k-1}, d_0)$.

[3] The number of cards in a stack would be revealed by the thickness of the stack.

We often omit the degree k and denote cyc_k as cyc.

$$(d_0, d_1, \ldots, d_{k-1}) \xrightarrow{\mathsf{Perm}\,\sigma} (d_{\sigma^{-1}(0)}, d_{\sigma^{-1}(1)}, \ldots, d_{\sigma^{-1}(k-1)}).$$
$$(d_0, d_1, \ldots, d_{k-1}) \xrightarrow{\mathsf{Perm\ cyc}} (d_1, \ldots, d_{k-1}, d_0).$$

Rotation. We define a *rotation operation* that takes as input a stack $d \in \mathcal{D}$ and outputs $\mathsf{rot}(d)$. Here, the rotated stack, $\mathsf{rot}(d)$, corresponds to subtracting 1 (modulo m) to the value of all front side cards $c \in \mathcal{F}$, and adding 1 from the value of all back side cards $c \in \mathcal{B}$ in the stack d. By $\mathsf{rot}^i(d)$ we denote the rotation operation applied i times. For $[\![x]\!]$ and a public number a, we can obtain $[\![x+a]\!]$ by applying $\mathsf{rot}^a([\![x]\!])$. We use the following notation.

$$d \xrightarrow{\mathsf{Rot}} \mathsf{rot}(d). \qquad\qquad d \xrightarrow{\mathsf{Rot}^i} \mathsf{rot}^i(d).$$

Flip. We define a *flip operation* that takes as input a card $c \in \mathcal{C}$ and outputs $\mathsf{flip}(c)$ which corresponds to the card c flipped around. For example, given the back side card $[\![x]\!]$, $\mathsf{flip}(x)$ corresponds to the front side x. When the input is a single back side card, we sometimes refer to the flip operation as open. We use the following notation.

$$c \xrightarrow{\mathsf{Flip}} \mathsf{flip}(c).$$

Shuffle. In this paper, we use two shuffles, called a cyclic shuffle and a rotation shuffle. Let a sequence $\boldsymbol{d} = (d_0, d_1, \cdots, d_{k-1}) \in \mathcal{D}^k$ satisfy $\mathsf{face}(d_0) = \mathsf{face}(d_1) = \cdots = \mathsf{face}(d_{k-1})$. We define a *cyclic shuffle* that takes as input a sequence $\boldsymbol{d}$ as above and outputs $\mathsf{cyc}^r(\boldsymbol{d})$ where r is uniformly chosen from $\mathbb{Z}_k$. (We assume that nobody knows the value r except when all parties are corrupted by an adversary.) Let a stack $d \in \mathcal{D}$ satisfy $\mathsf{face}(d) = \mathsf{face}(\mathsf{rot}(d)) = \cdots = \mathsf{face}(\mathsf{rot}^{m-1}(d))$. (Thus, the top of stack d is a back side card.) We define a *rotation shuffle* that takes as input a stack d as above and outputs $\mathsf{rot}^r(d)$ where r is uniformly chosen from $\mathbb{Z}_m$. (Similarly, we assume that nobody knows the value r.) As far as we know, all shuffles used in previous works can be expressed by the combination of the operations defined here[4]. As in previous works, we can securely operate a cyclic shuffle and a rotation shuffle[5].

$$(d_0, d_1, \cdots, d_{k-1}) \xrightarrow{\mathsf{CycShffl}} \mathsf{cyc}_k^r(d_0, d_1, \cdots, d_{k-1}).$$
$$d \xrightarrow{\mathsf{RotShffl}} \mathsf{rot}^r(d).$$

[4] The “cyclic shuffle” used in [2] and the random bisection cut proposed in [8] corresponds to one of our cyclic shuffles. Similarly, the shuffle used in [7] and the “rotation shuffle” used in [12] corresponds to one of the our rotation shuffles.

[5] We demonstrate how to securely obtain a cyclic shuffle. Let $P_0, \cdots, P_{n-1}$ be the parties participating in the protocol. P_0 chooses a uniformly random value $r_0 \in \mathbb{Z}_k$ and applies cyc^{r_0} to $\boldsymbol{d}$, and sends $\mathsf{cyc}^{r_0}(\boldsymbol{d})$ to P_1. Similarly, P_i receives $\boldsymbol{d'}$, chooses $r_i \in \mathbb{Z}_k$, and sends $\mathsf{cyc}^{r_i}(\boldsymbol{d'})$ to P_{i+1}. Finally, P_{n-1} outputs $\mathsf{cyc}^r(\boldsymbol{d})$ where $r = r_0 + \cdots + r_{n-1}$. Nobody knows the uniform random value r except when all parties are corrupted by an adversary assuming parties are *honest-but-curious*.

Composition/Decomposition. We define a *composition operation* that takes as input a pair of cards (c_0, c_1) where $c_0, c_1 \in \mathcal{C}$ and outputs a stack $c_0 \circ c_1 \in \mathcal{C}^2$. We define a *decomposition operation* as the inverse operation of a composition, i.e., it takes $c_0 \circ c_1 \in \mathcal{C}^2$ and outputs (c_0, c_1). Similarly, we define a *flip composition operation* that takes (c_0, c_1) where $c_0, c_1 \in \mathcal{C}$ and outputs $c_0 \circ \mathsf{flip}(c_1) \in \mathcal{C}^2$, and *flip decomposition operation* that is the inverse operation of a flip composition. We use the following notation.

$$(c_0, c_1) \xrightarrow{\mathsf{Comp}} c_0 \circ c_1. \qquad c_0 \circ c_1 \xrightarrow{\mathsf{Decomp}} (c_0, c_1).$$

$$(c_0, c_1) \xrightarrow{\mathsf{FComp}} c_0 \circ \mathsf{flip}(c_1). \qquad c_0 \circ c_1 \xrightarrow{\mathsf{FDecomp}} (c_0, \mathsf{flip}(c_1)).$$

Insert/Delete. We define an *insert operation* that takes as input a sequence $(d_0, d_1, \cdots, d_{k-1}) \in \mathcal{D}^k$ and outputs $(0, d_0, d_1, \cdots, d_{k-1}) \in \mathcal{D}^{k+1}$. Oppositely, we define a *delete operation* that takes as input a sequence $(d_0, d_1, \cdots, d_{k-1}) \in \mathcal{D}^k$ and outputs $(d_1, \cdots, d_{k-1}) \in \mathcal{D}^{k-1}$. Note that using permutation and rotation operations we can easily insert/delete any card at any position. We use the following notation.

$$(d_0, d_1, \cdots, d_{k-1}) \xrightarrow{\mathsf{Insert}} (0, d_0, d_1, \cdots, d_{k-1}).$$

$$(d_0, d_1, \cdots, d_{k-1}) \xrightarrow{\mathsf{Del}} (d_1, \cdots, d_{k-1}).$$

We have now defined all operations used in our protocols. However, we would like to apply these operations to a subsequence of the sequence. (For example, we want to apply a shuffle to a half of sequence.) For this reason, we use (naturally) extended operations that apply to a subsequence as follows.

$$(d_0, d_1, d_2, \cdots, d_{k-1}) \xrightarrow{\mathsf{RotShffl}\ \{0,1\}} (\mathsf{rot}^r(d_0), \mathsf{rot}^r(d_1), d_2, \cdots, d_{k-1}).$$

$$(c_0, c_1, d_2, \cdots, d_{k-1}) \xrightarrow{\mathsf{Flip}\ \{0,1\}} (\mathsf{flip}(c_0), \mathsf{flip}(c_1), d_2, \cdots, d_{k-1}).$$

$$(d_0, d_1, d_2, \cdots, d_{k-1}) \xrightarrow{\mathsf{Comp}\ \{1,2\}} (d_0, d_1 \circ d_2, d_3, \cdots, d_{k-1}).$$

$$(d_0, d_1, \cdots, d_{k-1}) \xrightarrow{\mathsf{Del}\ \{1,k-1\}} (d_0, d_2, d_3, \cdots, d_{k-2}).$$

We use $\mathcal{O}$ to denote the set of extended operations as above. Here, we stress that the shuffles are special operations[6]. We use $\mathcal{O}_S$ to denote the set of shuffles. The set of shuffles is a proper subset of the set of extended operations ($\mathcal{O}_S \subset \mathcal{O}$).

2.3 Model

Protocol. We define a protocol using regular m-sided polygon cards as follows.

Definition 2 (Protocol). *A protocol Π taking input $(x_0, x_1, \cdots, x_{n-1})$ is specified by $\langle (P_0, P_1, \cdots, P_{n-1}), (\mathsf{E}, \mathsf{E}'), \pi \rangle$, where P_i is a party who holds the secret*

[6] All operations except shuffles output a sequence in a deterministic way. However, shuffles output a sequence in a probabilistic way under a uniformly random value r.

value x_i, E and E' are the input and output encodings, and π is a transition function that takes as input a vector of faces of sequences, and outputs an element of $\mathcal{O} \cup \{\perp\}$. The protocol Π proceeds as follows.

1. *P_i submits $\mathsf{E}(x_i)$ in public. Let $\boldsymbol{c_0} := (\mathsf{E}(x_0), \mathsf{E}(x_1), \cdots, \mathsf{E}(x_{n-1}))$ be the initial sequence, and let $\boldsymbol{C}_0 := (\boldsymbol{c_0})$ be the initial vector of sequences.*
2. *Iteratively proceed as follows: For a vector $\boldsymbol{C_i} = (\boldsymbol{c_0}, \boldsymbol{c_1}, \cdots, \boldsymbol{c_i})$, apply the operation $\tau_i := \pi(\mathsf{face}(\boldsymbol{c_0}), \cdots, \mathsf{face}(\boldsymbol{c_i}))$ to the sequence $\boldsymbol{c_i}$, and obtain the sequence $\boldsymbol{c_i} + 1$.*
   - *When $\tau_i \notin \mathcal{O}_S$ (τ_i is not a shuffle operation), all parties publicly obtain $\boldsymbol{c_i}+1$ by applying τ_i to $\boldsymbol{c_i}$, i.e., $\boldsymbol{c_i} \xrightarrow{\tau_i} \boldsymbol{c_i}+1$. Note that all of these operations proceed in public.*
   - *When $\tau_i \in \mathcal{O}_S$ (τ_i is a shuffle operation), all parties invoke a shuffle protocol for $\boldsymbol{c_i}$, and obtain $\boldsymbol{c_i}+1$. In the shuffle protocol, each party P_i generates a uniformly random value r_i privately, and apply cyc^{r_i} or rot^{r_i} to the sequence. As a result, all parties obtain the sequence obtained from applying cyc^r or rot^r where $r := r_0 + r_1 + \cdots + r_{n-1}$. We use $\boldsymbol{c_i} \xrightarrow{\tau_i(r)} \boldsymbol{c_i}+1$ to denote a shuffle under the random value r.*
   - *When $\tau_i = \perp$, all of the parties output $\boldsymbol{c}_{\text{our}} := \boldsymbol{c_i}$ as an output sequence.*

For a protocol Π with input $\boldsymbol{x} = (x_0, x_1, \cdots, x_{n-1})$ we use $\mathsf{Trans}(\Pi, \boldsymbol{x})$ to denote the transcript of an execution of $\Pi(x)$, including the random values generated by the parties as part of the protocol:

$$\mathsf{Trans}(\Pi, \boldsymbol{x}) \to ((\mathsf{face}(\boldsymbol{c_0}), \mathsf{face}(\boldsymbol{c}_1), \cdots, \mathsf{face}(\boldsymbol{c_\ell})), \mathsf{face}(\boldsymbol{c}_{\text{out}}), \boldsymbol{r_0}, \boldsymbol{r}_1, \cdots, \boldsymbol{r_n} - 1)$$

where ℓ is the number of generated sequences, $\boldsymbol{c}_{\text{out}}$ is the output sequence $\boldsymbol{c}_{\text{out}}$, and $\boldsymbol{r_i}$ is a vector of random values[7] used by P_i.

Security. As in standard multi-party computation, we define security via a simulation-based security experiment. Intuitively, our notion of security captures that any set of corrupt users cannot learn anything about the secret input of honest users, expect for the output of the protocol. More specifically, for a protocol to be secure, we require that there exist an efficient simulator which can simulate the corrupt user's view of the protocol execution, without access to the honest user's input. We define a perfect security for a protocol Π as follows.

Definition 3 (Perfect Security). *Let Π be a protocol. We say that Π is* perfectly secure *if for any $\mathfrak{C} \subsetneq \mathbb{Z}_n$ there exists an efficient algorithm $\mathcal{S}$ such that for any $\boldsymbol{x} = (x_0, x_1, \cdots, x_{n-1})$ the outputs of the experiments $\mathsf{Exp}^0_{\Pi,\mathfrak{C}}(\boldsymbol{x})$ and $\mathsf{Exp}^1_{\Pi,\mathfrak{C},\mathcal{S}}(\boldsymbol{x})$ are distributed identically.*

$\underline{\mathsf{Exp}^0_{\Pi,\mathfrak{C}}(\boldsymbol{x})}$	$\underline{\mathsf{Exp}^1_{\Pi,\mathfrak{C},\mathcal{S}}(\boldsymbol{x})}$
$\mathsf{Trans}(\Pi, \boldsymbol{x})$	$\mathsf{Trans}(\Pi, \boldsymbol{x})$
$\to (\mathsf{face}(\boldsymbol{C}), \mathsf{face}(\boldsymbol{c}_{\text{out}}), \{\boldsymbol{r_i}\}_{i \in \mathbb{Z}_n})$;	$\to (\mathsf{face}(\boldsymbol{C}), \mathsf{face}(\boldsymbol{c}_{\text{out}}), \{\boldsymbol{r_i}\}_{i \in \mathbb{Z}_n})$;
$\mathsf{view} := (\mathsf{face}(\boldsymbol{C}), \{(x_i, \boldsymbol{r_i})\}_{i \in \mathfrak{C}})$;	$\mathcal{S}(\{x_i\}_{i \in \mathfrak{C}}, \mathsf{face}(\boldsymbol{c}_{\text{out}})) \to \mathsf{view}'$;
output view;	*output* view';

[7] When there are no shuffles in a transcript, the vector r_i should be an empty vector.

Efficiency. For a protocol Π, the *card complexity* and the *shuffle complexity* are defined as the worst-case number of cards and shuffles required in Π, respectively. We evaluate the efficiency of protocols with respect to the card complexity and shuffle complexity. Note that these complexities are not necessarily related, and cannot be compared directly. Furthermore, since the shuffle operation is the only randomized operation, the shuffle complexity can also be seen as a measure for demanding a protocol is with respect to random number generation.

3 Efficient Protocols Using Regular Polygon Cards

In this section, using properties of regular m-sided polygon cards, we construct protocols for linear functions over $\mathbb{Z}_m$. Linear functions are all functions that can be expressed as a composition of multiplications by a constant and additions. In particular, we construct four protocols as follows.

- Addition Protocol : $([\![x_0]\!], [\![x_1]\!]) \xrightarrow{\mathsf{Add}} [\![x_0 + x_1]\!]$.
- Subtraction Protocol : $([\![x_0]\!], [\![x_1]\!]) \xrightarrow{\mathsf{Sub}} [\![x_1 - x_0]\!]$.
- Copy Protocol : $[\![x]\!] \xrightarrow{\mathsf{Copy}\ k} (\underbrace{[\![x]\!], [\![x]\!], \cdots, [\![x]\!]}_{k})$.
- Multiplication by Constant Protocol : $[\![x]\!] \xrightarrow{\mathsf{Mult}\ a} [\![ax]\!]$.

Composing them, we can securely compute an arbitrary linear function over $\mathbb{Z}_m$.

3.1 Addition, Subtraction, and Copy Protocols

We construct an *addition protocol* over $\mathbb{Z}_m$ using regular m-sided polygon cards. It takes as inputs two encodings $[\![x_0]\!]$ and $[\![x_1]\!]$, and outputs an encoding of the sum $[\![x_0 + x_1]\!]$. Our idea is simple. Firstly, using a rotation shuffle, we obtain two encodings $[\![x_0 + r]\!]$ and $[\![x_1 - r]\!]$, and open the former to learn the value $\epsilon := x_0 + r$. Since ϵ is masked by the random value r, ϵ reveals no information about x_0. Once ϵ is opened in public, we can easily obtain $[\![x_0 + x_1]\!]$ from $[\![x_1 - r]\!]$ by applying an ϵ-times rotation. The most important part in our addition protocol is generating the rotation and the inverse rotation $[\![x_0 + r]\!]$ and $[\![x_1 - r]\!]$. You can see the demonstration movie (https://youtu.be/9Tid6X-9r-c).

Addition Protocol (Add)

Secret Information : $(x_0, x_1) \in (\mathbb{Z}_m, \mathbb{Z}_m)$.
Input : $([\![x_0]\!], [\![x_1]\!])$.
Output$[\![x_0 + x_1 \bmod m]\!]$. (We omit "mod m" in the description.)

1. Apply a flip composition to $([\![x_0]\!], [\![x_1]\!])$.

$$([\![x_0]\!], [\![x_1]\!]) \xrightarrow{\mathsf{FComp}} [\![x_0]\!] \circ \mathsf{flip}([\![x_1]\!]).$$

2. Apply a rotation shuffle to $[\![x_0]\!] \circ \mathsf{flip}([\![x_1]\!])$.

$$\begin{aligned}([\![x_0]\!] \circ \mathsf{flip}([\![x_1]\!])) &\xrightarrow{\mathsf{RotShffl}} \mathsf{rot}^r([\![x_0]\!]) \circ \mathsf{rot}^r(\mathsf{flip}([\![x_1]\!])) \\ &= [\![x_0 + r]\!] \circ \mathsf{flip}([\![x_1 - r]\!]).\end{aligned}$$

3. Apply a flip decomposition to $[\![x_0 + r]\!] \circ \mathsf{flip}([\![x_1 - r]\!])$. Open the 1st card $[\![x_0 + r]\!]$, and learn a value $\epsilon := x_0 + r \bmod m$ publicly.

$$[\![x_0 + r]\!] \circ \mathsf{flip}([\![x_1 - r]\!]) \xrightarrow{\mathsf{FDecomp}} ([\![x_0 + r]\!], [\![x_1 - r]\!]) \xrightarrow{\mathsf{Open}\ \{0\}} (\epsilon, [\![x_1 - r]\!]).$$

4. Delete the front card ϵ. Apply ϵ-times rotation to $[\![x_1 - r]\!]$.

$$(\epsilon, [\![x_1 - r]\!]) \xrightarrow{\mathsf{Del}\ \{0\}} [\![x_1 - r]\!] \xrightarrow{\mathsf{Rot}\ \epsilon} [\![x_0 + x_1]\!].$$

Theorem 1. *The above addition protocol is perfectly secure.*

Proof. The faces of the sequence in the above addition protocol are the following.

$$(\times, \times) \xrightarrow{\mathsf{FComp}} (\times^2) \xrightarrow{\mathsf{RotShffl}} (\times^2) \xrightarrow{\mathsf{FDecomp}} (\times, \times) \xrightarrow{\mathsf{Open}} (``\epsilon", \times) \xrightarrow{\mathsf{Del}} (\times) \xrightarrow{\mathsf{Rot}} (\times).$$

In the sequences, $\epsilon = x_0 + r$ is a uniformly random value in $\mathbb{Z}_m$ since r is a uniformly random value in $\mathbb{Z}_m$. For any subset $\mathfrak{C} \subsetneq \mathbb{Z}_n$, the simulator $\mathcal{S}$ is constructed as follows. $\mathcal{S}$ takes $\{x_i\}_{i \in \mathfrak{C}}$ and a face of the output sequence $(\times)$ as inputs, chooses uniformly random r'_i and $\epsilon' \in \mathbb{Z}_m$, and outputs the following.

$$(((\times, \times), (\times^2), (\times^2), (\times, \times), (``\epsilon'", \times), (\times), (\times)), \{x_i, r'_i\}_{i \in \mathfrak{C}}).$$

This is the same as the real distribution. Therefore, our addition protocol Add is perfectly secure. □

In the addition protocol, if we use $[\![x_0]\!] \circ \mathsf{flip}([\![x_1]\!]) \circ \mathsf{flip}([\![x_2]\!]) \cdots \circ \mathsf{flip}([\![x_{k-1}]\!])$ instead of $[\![x_0]\!] \circ \mathsf{flip}([\![x_1]\!])$, then it is possible to apply an x_0-addition operation to $k-1$ encodings by using a rotation shuffle (Single Instruction Multiple Data, SIMD operation).

$$([\![x_0]\!], [\![x_1]\!], \cdots, [\![x_{k-1}]\!]) \xrightarrow{\mathsf{Add}} ([\![x_0 + x_1]\!], \cdots, [\![x_0 + x_{k-1}]\!]).$$

Using this property, we immediately obtain a copy protocol by applying our addition protocol to $[\![x]\!]$ and $[\![0]\!]$s, i.e., $([\![x]\!], [\![0]\!], \cdots, [\![0]\!]) \xrightarrow{\mathsf{Add}} ([\![x]\!], \cdots, [\![x]\!])$. We use the following notation for our k-copy protocol.

$$[\![x]\!] \xrightarrow{\mathsf{Copy}\ k} (\underbrace{[\![x]\!], [\![x]\!], \cdots, [\![x]\!]}_{k}).$$

In the addition protocol, if we use $[\![x_0]\!] \circ [\![x_1]\!]$ instead of $[\![x_0]\!] \circ \mathsf{flip}([\![x_1]\!])$, then we obtain a subtraction protocol. Similarly, it is possible to operate a SIMD operation in our subtraction as follows.

$$([\![x_0]\!], [\![x_1]\!], \cdots, [\![x_{k-1}]\!]) \xrightarrow{\mathsf{Sub}} ([\![x_1 - x_0]\!], \cdots, [\![x_{k-1} - x_0]\!]).$$

Clearly, it is possible to apply a SIMD operation for $[\![x_0]\!]$-addition, $[\![x_0]\!]$-subtraction, and copy for $[\![x_0]\!]$. For example, the following operation can be obtained from a rotation shuffle.

$$([\![x_0]\!], [\![x_1]\!], [\![x_2]\!], [\![x_3]\!]) \to ([\![x_1 + x_0]\!], [\![x_2 - x_0]\!], [\![x_3 - x_0]\!], [\![x_0]\!], [\![x_0]\!]).$$

3.2 Protocol for Multiplication by a Constant

A protocol for multiplication by a constant, that takes an encoding $[\![x]\!]$ and a constant $a \in \mathbb{Z}_m$ as inputs and outputs $[\![ax]\!]$, can be constructed by just applying our addition protocol a times. Using a binary representation $a = a_0 + 2a_1 + \cdots + 2^{\ell-1}a_{\ell-1}$, it is widely known how to reduce the complexity for a multiplication. (We refer to this as the *binary method.*) In our model, we can apply the binary method and obtain a protocol using only $O(\log a)$-times rotation shuffles as follows.

1. Let $a \in \mathbb{Z}_m$ denote a constant represented by $a = a_0 + 2a_1 + \cdots + 2^{\ell-1}a_{\ell-1}$ where $a_i \in \{0, 1\}$.
2. Repeat the following from $i = 0$ to $\ell - 1$.
   (a) If $a_i = 0$, then apply our 2-copy protocol to $[\![2^i x]\!]$, otherwise apply our 3-copy protocol to $[\![2^i x]\!]$.
   (b) For two encodings of $[\![2^i x]\!]$, apply our addition protocol and obtain an encoding of $[\![2^{i+1} x]\!]$.
3. For all encodings generated as above, apply our addition protocol and obtain an encoding of $[\![ax]\!]$.

Let ℓ_a be the number of a_i that satisfies $a_i = 1$. The shuffle complexity of the above protocol is $(2\lceil \log_2 a \rceil + \ell_a - 1)$ rotation shuffles.

In the rest of this section, we show that it is possible to construct a multiplication by constant protocol whose shuffle complexity is only $(\lceil \log_2 a \rceil + 1)$ rotation shuffles. The basic idea is to use a SIMD operation of our addition protocol. Our multiplication by constant protocol is denoted by $[\![x]\!] \xrightarrow{\mathsf{Mult}\ a} [\![ax]\!]$.

Multiplication by Constant Protocol (Mult)

Secret Information : $x \in \mathbb{Z}_m$.
Input : $[\![x]\!]$.
Output$[\![ax]\!]$.
Let $\ell = \lceil \log_2 a \rceil$ and $a - 1 = \sum_{j=0}^{\ell-1} 2^j \cdot b_j$ where $b_j \in \{0, 1\}$. Note that we use a binary representation of $a - 1$, while the standard binary method using that of a.

1. Invoke our $(\ell + 1)$-copy protocol to $[\![x]\!]$.

$$[\![x]\!] \xrightarrow{\mathsf{Copy}\ \ell+1} \underbrace{([\![x]\!], \cdots, [\![x]\!])}_{\ell+1}.$$

2. Repeat the following operation for $i = 0, 1, \cdots, \ell - 1$ the following operation. For clarity, we first demonstrate the 0th step and then the i-th step. In the 0th step, invoke our addition protocol Add as follows: if $b_0 = 1$, then add the leftmost $[\![x]\!]$ to all others $[\![x]\!]$, otherwise add the leftmost $[\![x]\!]$ to all others $[\![x]\!]$ except from the rightmost $[\![x]\!]$.

$$(\underbrace{[\![x]\!], \cdots, [\![x]\!]}_{\ell}, [\![x]\!]) \xrightarrow{\mathsf{Add}} \begin{cases} (\underbrace{[\![2x]\!], \cdots, [\![2x]\!]}_{\ell-1}, [\![x]\!]) & \text{if } b_0 = 0. \\ (\underbrace{[\![2x]\!], \cdots, [\![2x]\!]}_{\ell-1}, [\![2x]\!]) & \text{if } b_0 = 1. \end{cases}$$

In i-th step ($i > 0$), the current sequence is $\boldsymbol{s} = (\overbrace{[\![w]\!], \cdots, [\![w]\!]}^{\ell-i}, [\![w_i]\!])$ where $w = 2^i x$, and $w_i = (\sum_{j=0}^{i-1} 2^j b_j + 1)x$. If $b_i = 1$, then add $[\![w]\!]$ to all others. If $b_i = 0$, then add $[\![w]\!]$ to all others except from $[\![w_i]\!]$.

$$(\underbrace{[\![w]\!], \cdots, [\![w]\!]}_{\ell-i}, [\![w_i]\!]) \xrightarrow{\mathsf{Add}} \begin{cases} (\underbrace{[\![2w]\!], \cdots, [\![2w]\!]}_{\ell-i-1}, [\![w_i]\!]) & \text{if } b_i = 0. \\ (\underbrace{[\![2w]\!], \cdots, [\![2w]\!]}_{\ell-i-1}, [\![w_i + w]\!]) & \text{if } b_i = 1. \end{cases}$$

3. Finally, the current sequence is a card $[\![w_\ell]\!]$, where $w_\ell = (\sum_{j=0}^{\ell-1} 2^j b_j + 1)x = ax$. The output is the card $[\![w_\ell]\!]$.

Theorem 2. *The above multiplication protocol is perfectly secure.*

Proof. The faces of the sequence in the above multiplication protocol are the following.

$$(\times) \xrightarrow{\mathsf{Copy}} (\underbrace{\times, \times, \cdots, \times}_{\ell+1}) \xrightarrow{\mathsf{Add}} (\underbrace{\times, \times, \cdots, \times}_{\ell}) \xrightarrow{\mathsf{Add}} \cdots \xrightarrow{\mathsf{Add}} (\times, \times) \xrightarrow{\mathsf{Add}} (\times).$$

For any subset $\mathfrak{C} \subsetneq \mathbb{Z}_n$, the simulator $\mathcal{S}$ is constructed as follows. $\mathcal{S}$ takes $\{x_i\}_{i \in \mathfrak{C}}$ and the face of the output sequence $(\times)$ as inputs, chooses uniformly random $r'_{i,j}$ and $\epsilon'_j \in \mathbb{Z}_m$ (for $i \in \mathfrak{C}$ and $j \in \{0, 1, \cdots, \ell\}$), and outputs $(\mathsf{face}(\boldsymbol{C'}), \{x_i, r'_{i,0}, r'_{i,1}, \cdots, r'_{i,\ell}\}_{i \in \mathfrak{C}})$. (Note that $\mathsf{face}(\boldsymbol{C'})$ can be easily generated from ϵ'_j and the transition function π.) This is the same as the real distribution. Therefore, our multiplication protocol Mult_a is perfectly secure. □

4 Efficient Protocols Using Oblivious Conversion

In this section, we construct protocols for any function $f : \bigotimes_{i=0}^{n-1} \mathbb{Z}_{m_i} \to \mathbb{Z}_{m_n}$ with small shuffle complexity using a new protocol, an *oblivious conversion.* A protocol based on oblivious conversion has a small shuffle complexity, but use a large number of cards. In general, there is a trade-off between a specific construction and a generic (oblivious conversion based) construction, in terms of the shuffle complexity and the number of cards. (For example, the specific multiplication by $a \in \mathbb{Z}_m$ (Sect. 3.2) uses $\lceil \log_2 a \rceil + 1$ rotation shuffles and $\lceil \log_2 a \rceil + 2$ cards while a multiplication protocol based on oblivious conversion uses 1 cyclic shuffle and $2m$ cards (Example 1).) Our oblivious conversion can be applied to general encodings (see Corollary 1). Thus, for general encodings we can construct protocols for any functions with small shuffle complexity (see Corollary 2).

For two finite cyclic groups $\mathbb{Z}_{m_0}, \mathbb{Z}_{m_1}$, let f be a function s.t. $f : \mathbb{Z}_{m_0} \to \mathbb{Z}_{m_1}$, and let $\mathsf{E}_0, \mathsf{E}_1$ be encodings on $\mathbb{Z}_{m_0}, \mathbb{Z}_{m_1}$, respectively. An *oblivious conversion* is defined by a protocol, that takes $\mathsf{E}_0(x), \mathsf{E}_1(f(0)), \mathsf{E}_1(f(1)), \cdots, \mathsf{E}_1(f(m_0 - 1))$ as inputs, and outputs $\mathsf{E}_1(f(x))$. In our definition, an oblivious transfer can be seen as an oblivious conversion. Indeed, if two parties P_0 and P_1 plays a receiver and a sender, i.e., P_0 chooses x, and P_1 chooses f, then the oblivious conversion is equal to an oblivious transfer.

Firstly, we construct an oblivious conversion for $f : \mathbb{Z}_m \to \mathbb{Z}_m$, where the input and output encodings are the standard encoding $[\![\cdot]\!]$. You can see the demonstration movie (https://youtu.be/hlAetm66iRU).

Oblivious Conversion

Secret Information : $x \in \mathbb{Z}_m$.
Input : $([\![x]\!], \boldsymbol{f})$ where $\boldsymbol{f} = ([\![f(0)]\!], [\![f(1)]\!], \cdots, [\![f(m-1)]\!])$.
Output : $[\![f(x)]\!]$.

1. Invoke our m-copy protocol for $[\![x]\!]$, apply a $(i-1)$-times rotation to i-th card from the left ($i = 1, 2, \cdots, m$), and obtain $\boldsymbol{x} = ([\![x]\!], [\![x-1]\!], [\![x-2]\!], \cdots, [\![x-(m-1)]\!])$.

$$([\![x]\!], \boldsymbol{f}) \xrightarrow{\mathsf{Copy}\ m} (\underbrace{[\![x]\!], \cdots, [\![x]\!]}_{m}, \boldsymbol{f}) \xrightarrow{\mathsf{Rot}} (\boldsymbol{x}, \boldsymbol{f}).$$

   We see the sequence as a matrix as follows.

$$\begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{f} \end{pmatrix} = \begin{pmatrix} [\![x]\!] & [\![x-1]\!] & \cdots & [\![0]\!] & \cdots & [\![x-(m-1)]\!] \\ [\![f(0)]\!] & [\![f(1)]\!] & \cdots & [\![f(x)]\!] & \cdots & [\![f(m-1))]\!] \end{pmatrix}.$$

2. Apply a composition operation to each column and make a sequence $\boldsymbol{w} := (w_0, w_1, \cdots, w_{m-1})$ where $w_i = [\![x-i]\!] \circ [\![f(i)]\!]$. Apply a cyclic shuffle to the sequence, and then decompose it.

$$\begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{f} \end{pmatrix} \xrightarrow{\mathsf{Comp}} \boldsymbol{w} \xrightarrow{\mathsf{CycShffl}} \mathsf{cyc}^r(\boldsymbol{w}) \xrightarrow{\mathsf{Decomp}} \begin{pmatrix} \mathsf{cyc}^r(\boldsymbol{x}) \\ \mathsf{cyc}^r(\boldsymbol{f}) \end{pmatrix}.$$

3. Open the first card of $\mathsf{cyc}^r(\boldsymbol{x})$, and learn the value $\epsilon := x - r \bmod m$. Delete the top column in the matrix.

4. Apply an ϵ-times cyclic permutation to the sequence.

$$\mathsf{cyc}^r(\boldsymbol{f}) \xrightarrow{\mathsf{Perm\ cyc}^\epsilon} \mathsf{cyc}^x(\boldsymbol{f}).$$

5. The 1st card of $\mathsf{cyc}^x(\boldsymbol{f})$ is $[\![f(x)]\!]$, this is the output.

$$([\![f(x)]\!], [\![f(x+1)]\!], \cdots, [\![f(x+m-1)]\!]) \xrightarrow{\mathsf{Del}} [\![f(x)]\!].$$

Theorem 3. *The above oblivious conversion is perfectly secure.*

Proof. In the case of $m = 4$, the faces of the sequence in the above protocol are the following.

$$(\times,\times,\times,\times,\times) \xrightarrow{\mathsf{Copy}} (\times,\times,\times,\times,\times,\times,\times,\times) \xrightarrow{\mathsf{Rot}} (\times,\times,\times,\times,\times,\times,\times,\times) \longrightarrow$$
$$\begin{pmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \end{pmatrix} \xrightarrow{\mathsf{Comp}} (\times^2,\times^2,\times^2,\times^2) \xrightarrow{\mathsf{CycShffl}} (\times^2,\times^2,\times^2,\times^2) \xrightarrow{\mathsf{Decomp}} \begin{pmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \end{pmatrix}$$
$$\xrightarrow{\mathsf{Open}} \begin{pmatrix} \text{``}\epsilon\text{''} & \times & \times & \times \\ \times & \times & \times & \times \end{pmatrix} \xrightarrow{\mathsf{Del}} (\times,\times,\times,\times) \xrightarrow{\mathsf{cyc}^\epsilon} (\times,\times,\times,\times) \xrightarrow{\mathsf{Del}} (\times).$$

For any subset $\mathfrak{C} \subsetneq \mathbb{Z}_n$, $\mathcal{S}$ takes $\{x_i\}_{i\in\mathfrak{C}}$ and the face of the output sequence $(\times)$ as inputs, chooses uniformly random $r'_i \in \mathbb{Z}_m$ (for $i \in \mathfrak{C}$) and $\epsilon' \in \mathbb{Z}_m$, and outputs $(\mathsf{face}(\boldsymbol{C'}), \{x_i, r'_i\}_{i\in\mathfrak{C}})$. (Note that $\mathsf{face}(\boldsymbol{C'})$ can be easily generated from ϵ' and the transition function π.) This is the same as the real distribution. Therefore, our oblivious conversion is perfectly secure. $\square$

In above protocol, using an encoding E_1 instead of $[\![\cdot]\!]$, we can obtain an oblivious conversion protocol that takes $\boldsymbol{f} = (\mathsf{E}_1(f(0)), \cdots, \mathsf{E}_1(f(m-1)))$ as inputs. Similarly, if an encoding E_0 can execute Step 1 in the protocol, then we can use $\mathsf{E}_0(x)$ instead of $[\![x]\!]$ as input. This is formally stated in the following.

Corollary 1. *For cyclic groups $\mathbb{Z}_{m_0}, \mathbb{Z}_{m_1}$, let $f : \mathbb{Z}_{m_0} \to \mathbb{Z}_{m_1}$ be a function and $\mathsf{E}_0, \mathsf{E}_1$ be encodings on X, Y such that there exists integers k_0, k_1 that satisfies for all $x \in \mathbb{Z}_{m_0}$ $\mathsf{face}(\mathsf{E}_0(x)) = \times^{k_0}$ and $\mathsf{face}(\mathsf{E}_1(f(x))) = \times^{k_1}$. If the encoding E_0 supports the computation of $(\mathsf{E}_0(x), \mathsf{E}_0(x-1), \cdots, \mathsf{E}_0(x-(m_0-1)))$ from $\mathsf{E}_0(x)$ by applying ℓ shuffles, then there exists an oblivious conversion for f with only $\ell + 1$ shuffles.*

This can be easily proven from the construction of our oblivious conversion.

Example 1 (Multiplication by a Constant Protocol). *For a function $f : \mathbb{Z}_m \to \mathbb{Z}_m$ defined as $f(x) = ax$, and encodings $\mathsf{E}_0(\cdot) = \mathsf{E}_1(\cdot) = [\![\cdot]\!]$, the oblivious conversion for f is a protocol of multiplication by a constant a.*

Example 2 (Square Protocol). *For a function $f : \mathbb{Z}_m \to \mathbb{Z}_m$ defined as $f(x) = x^2$, and encodings $\mathsf{E}_0(\cdot) = \mathsf{E}_1(\cdot) = [\![\cdot]\!]$, the oblivious conversion for f is a square protocol.*

Example 3 (Modulus Switch). *For a function $f : \mathbb{Z}_{m_0} \to \mathbb{Z}_{m_1}$ defined as $f(x) = (x \bmod m_1)$, and encodings $\mathsf{E}_0(x) = [\![x]\!]$ (the natural encoding of regular m_0-sided polygon cards) $\mathsf{E}_1(y) = [\![y]\!]$ (the natural encoding of regular m_1-sided polygon cards), the oblivious conversion for f is a modulus switch protocol.*

For all $x \in \mathbb{Z}_{m_0}$, if an encoding E_0 satisfies $\mathsf{cyc}(\mathsf{E}_0(x)) = \mathsf{E}_0(x - 1 \bmod m_0)$, then we can obtain an oblivious conversion which has a lower shuffle complexity. For such an encoding, there exists a constant k which satisfies $\mathsf{E}(x) \in \mathcal{B}^{km}$, since $\mathsf{cyc}^m(\mathsf{E}(x)) = \mathsf{E}(x - m) = \mathsf{E}(x)$. Thus, we can use $\mathsf{E}(x)$ instead of $\boldsymbol{x}$ in Step 2. As a result, we obtain an oblivious conversion using only a cyclic shuffle.

Example 4 (Encodings for Small Shuffle Complexity). *For $\mathbb{Z}_m$, let $\mathsf{E}_A(x) = ([\![x]\!], [\![x-1]\!], \cdots, [\![x-(m-1)]\!]) \in \mathcal{B}^m$, and $\mathsf{E}_B(x) = ([\![y_0]\!], [\![y_1]\!], \cdots, [\![y_{m-1}]\!]) \in \mathcal{B}^m$ where $y_j = 0$ for $(j \neq x)$ and $y_x = 1$. And then E_A and E_B satisfy $\forall x\big[\mathsf{cyc}_k(\mathsf{E}(x)) = \mathsf{E}(x-1)\big]$. Furthermore, let E_C be an encoding which have redundancy, s.t. $\mathsf{E}_C(x) = (\mathsf{E}_A(x), \mathsf{E}_A(x)) \in \mathcal{B}^{2m}$. Then E_C satisfies $\forall x\big[\mathsf{cyc}_{2k}(\mathsf{E}_C(x)) = \mathsf{E}_C(x-1)\big]$.*

MPC for Small Shuffle Complexity. It is possible to construct a secure MPC for any function using only our oblivious conversion. We use $(\alpha_i)_{i=0}^{m-1}$ to denote the sequence $(\alpha_0, \alpha_1, \cdots, \alpha_{m-1})$. For a function $f : \bigotimes_{i=0}^{n-1} \mathbb{Z}_{m_i} \to \mathbb{Z}_{m_n}$ and an output encoding E_n, we first define a sequence $\boldsymbol{f}$ as follows.

$$\begin{aligned}
\boldsymbol{f}^{(x_0,\cdots,x_{n-3},x_{n-2})} &:= (\mathsf{E}_n(f(x_0,\cdots,x_{n-2},i)))_{i=0}^{m_{n-1}-1}\\
\boldsymbol{f}^{(x_0,\cdots,x_{n-3})} &:= (\boldsymbol{f}^{(x_0,\cdots,x_{n-3},i)})_{i=0}^{m_{n-2}-1}\\
&\vdots\\
\boldsymbol{f}^{(x_0)} &:= (\boldsymbol{f}^{(x_0,i)})_{i=0}^{m_1-1}\\
\boldsymbol{f} &:= (\boldsymbol{f}^{(i)})_{i=0}^{m_0-1}
\end{aligned}$$

Let $\mathsf{E}_0, \mathsf{E}_1, \cdots, \mathsf{E}_{n-1}$ be encodings over $\mathbb{Z}_{m_0}, \mathbb{Z}_{m_1}, \cdots, \mathbb{Z}_{m_{n-1}}$, respectively. Given encodings of inputs $\{\mathsf{E}_i(x_i)\}_{i \in \mathbb{Z}_n}$ and $\boldsymbol{f}$, we can obtain the output sequence $\mathsf{E}_n(f(x_0, \cdots, x_{n-1}))$ as follows.

$$\begin{aligned}
&(\mathsf{E}_0(x_0), \boldsymbol{f}) \xrightarrow{\mathsf{OC}} \boldsymbol{f}^{(x_0)}\\
&(\mathsf{E}_1(x_1), \boldsymbol{f}^{(x_0)}) \xrightarrow{\mathsf{OC}} \boldsymbol{f}^{(x_0,x_1)}\\
&\qquad\vdots\\
&(\mathsf{E}_{n-2}(x_{n-2}), \boldsymbol{f}^{(x_0,\cdots,x_{n-3})}) \xrightarrow{\mathsf{OC}} \boldsymbol{f}^{(x_0,\cdots,x_{n-3},x_{n-2})}\\
&(\mathsf{E}_{n-1}(x_{n-1}), \boldsymbol{f}^{(x_0,\cdots,x_{n-3},x_{n-2})}) \xrightarrow{\mathsf{OC}} \mathsf{E}_n(f(x_0,\cdots,x_{n-1}))
\end{aligned}$$

where $\xrightarrow{\mathsf{OC}}$ denotes applying the oblivious conversion.

Corollary 2. *For cyclic groups $\mathbb{Z}_{m_0}, \mathbb{Z}_{m_1}, \cdots, \mathbb{Z}_{m_n}$, let $f : \bigotimes_{i=0}^{n-1} \mathbb{Z}_{m_i} \to \mathbb{Z}_{m_n}$ be a function and let E_i $(i \in \{0, 1, \cdots, n-1\})$ and E_n be encodings on $\mathbb{Z}_{m_i}$ and*

$\mathbb{Z}_{m_n}$ such that there exists integers $k_0, k_1, \cdots, k_n$ that satisfies for all $x_i \in \mathbb{Z}_{m_i}$ $\mathsf{face}(\mathsf{E}_i(x_i)) = \times^{k_i}$ $(i \in \{0, 1, \cdots, n-1\})$ and $\mathsf{face}(\mathsf{E}_n(f(x_0, x_1, \cdots, x_{n-1}))) = \times^{k_n}$. If for all $\mathbb{Z}_{m_i}$ $(i \in \{0, 1, \cdots, n-1\})$ the encoding E_i supports the computation of $(\mathsf{E}_i(x), \mathsf{E}_i(x-1), \cdots, \mathsf{E}_i(x-(m_i-1)))$ from $\mathsf{E}(x)$ by applying ℓ_i shuffles, then there exists a card-based protocol for f with only $\sum_{i=0}^{n-1}(\ell_i + 1)$ shuffles. In particular, if each E_i satisfies $\forall x \in \mathbb{Z}_{m_i}\big[\mathsf{cyc}^r(\mathsf{E}_i(x)) = \mathsf{E}_i(x-r)\big]$ $(i \in \{0, 1, \cdots, n-1\})$, then there exists a card-based protocol for f with only n cyclic shuffles.

This corollary can be easily proven from the above discussion.

5 Efficient Voting Protocol for Multiple Candidates

Using our oblivious conversion (Sect. 4), it is possible to construct a protocol for any n-ary function with n cyclic shuffles. Since protocols based on oblivious conversion are generic constructions, they require a large number of cards. Therefore, it might be possible to construct a more efficient protocol in terms of both the shuffle complexity and the number of cards by considering a protocol tailored to a specific functionality. In this section, we construct an efficient voting protocol for ℓ candidates and n voters. Assume each voter P_i holds $x_i \in \mathbb{Z}_\ell$, i.e., P_i supports the x_i-th candidate. For $x \in \mathbb{Z}_\ell$, let $\mathsf{E}_v(x)$ be an encoding for voting as follows[8]

$$\mathsf{E}_v(x) := \mathsf{cyc}_\ell^{-x}(\underbrace{[\![1]\!], [\![0]\!], [\![0]\!], \cdots, [\![0]\!]}_{\ell}).$$

Clearly, this encoding satisfies $\forall x\big[\mathsf{cyc}_\ell(\mathsf{E}_v(x)) = \mathsf{E}_v(x - 1 \bmod \ell)\big]$. Our voting protocol takes n encodings $\mathsf{E}_v(x_0), \mathsf{E}_v(x_1), \cdots, \mathsf{E}_v(x_{n-1})$ as inputs, and outputs $\boldsymbol{y} = ([\![y_0]\!], \cdots, [\![y_{\ell-1}]\!])$ where y_j is the number of votes for the j-th candidate. (Note that our voting protocol outputs encodings of the number of votes, thus we can apply an arbitrary functionality to the outputs.) The shuffle complexity of our voting protocol is $n+1$ cyclic shuffles while a protocol using our oblivious conversion has only n cyclic shuffles. However, the number of cards is only $(n+2)\ell$ while the protocol using our oblivious conversion needs $O(\ell^n)$.

For the simplicity, we show a voting protocol with 2 voters. It is easy to extend this to a voting protocol with n voters for an arbitrary n.

Voting Protocol with 2 Voters and ℓ Candidates

Secret Information : $(x_0, x_1) \in \mathbb{Z}_\ell^2$.
Input : $\mathsf{E}_v(x_0), \mathsf{E}_v(x_1)$.
Output : $([\![y_0]\!], \cdots, [\![y_{\ell-1}]\!])$ where $y_i = |\{j | x_j = i\}|$.

[8] The encoding E_v is just equal to E_B (Sect. 4, Example 4).

1. We deal with the input sequence $(\mathsf{E}_v(x_0), \mathsf{E}_v(x_1))$ as a matrix as bellow. Insert $\mathsf{E}_v(0) = ([\![1]\!], [\![0]\!], \cdots, [\![0]\!])$ and all-zero sequence $\mathbf{0} = ([\![0]\!], \cdots, [\![0]\!])$.

$$\begin{pmatrix} \mathsf{E}_v(x_0) \\ \hline \mathsf{E}_v(x_1) \end{pmatrix} \xrightarrow{\mathsf{Insert}} \begin{pmatrix} \mathsf{E}_v(x_0) \\ \hline \mathsf{E}_v(x_1) \\ \hline [\![1]\!]\ [\![0]\!]\ \cdots\ [\![0]\!]\ [\![0]\!] \\ \hline [\![0]\!]\ [\![0]\!]\ \cdots\ [\![0]\!]\ [\![0]\!] \end{pmatrix} = \begin{pmatrix} \mathsf{E}_v(x_0) \\ \hline \mathsf{E}_v(x_1) \\ \hline \mathsf{E}_v(0) \\ \hline \mathbf{0} \end{pmatrix}.$$

2. Apply a composition to each column and make a sequence $\boldsymbol{w_0} = (w_{0,0}, w_{0,1}, \cdots, w_{0,\ell-1})$. (Note that each $w_{0,i}$ contains 4 cards.) Apply a cyclic shuffle to $\boldsymbol{w_0}$, and decompose it.

$$\begin{pmatrix} \mathsf{E}_v(x_0) \\ \hline \mathsf{E}_v(x_1) \\ \hline \mathsf{E}_v(0) \\ \hline \mathbf{0} \end{pmatrix} \xrightarrow{\mathsf{Comp}} \boldsymbol{w_0} \xrightarrow{\mathsf{CycShffl}} \mathsf{cyc}^{r_0}(\boldsymbol{w_0}) \xrightarrow{\mathsf{Decomp}} \begin{pmatrix} \mathsf{E}_v(x_0 - r_0) \\ \hline \mathsf{E}_v(x_1 - r_0) \\ \hline \mathsf{E}_v(-r_0) \\ \hline \mathsf{cyc}^{r_0}(\mathbf{0}) \end{pmatrix}.$$

3. Open the first row and learn the value $\epsilon_0 := x_0 - r_0$, and delete the first row. For the bottom sequence $\mathsf{cyc}^{r_0}(\mathbf{0})$, apply a rotation rot to ϵ_0-th card from the right, i.e., x_0-th card of $\mathbf{0}$. (Note that we refer to the rightmost card as the "0th" card.) Let $\mathsf{cyc}^{r_0}(\boldsymbol{z_0})$ be the bottom sequence.
4. Apply a composition to each column and make a sequence $\boldsymbol{w_1} = (w_{1,0}, w_{1,1}, \cdots, w_{1,\ell-1})$. (Note that each $w_{1,i}$ contains 3 cards.) Apply a cyclic shuffle to $\boldsymbol{w_1}$, and decompose it.

$$\begin{pmatrix} \mathsf{E}_v(x_1 - r_0) \\ \hline \mathsf{E}_v(-r_0) \\ \hline \mathsf{cyc}^{r_0}(\boldsymbol{z_0}) \end{pmatrix} \xrightarrow{\mathsf{Comp}} \boldsymbol{w_1} \xrightarrow{\mathsf{CycShffl}} \mathsf{cyc}^{r_1}(\boldsymbol{w_1}) \xrightarrow{\mathsf{Decomp}} \begin{pmatrix} \mathsf{E}_v(x_1 - r) \\ \hline \mathsf{E}_v(-r) \\ \hline \mathsf{cyc}^{r}(\boldsymbol{z_0}) \end{pmatrix}.$$

   where $r = r_0 + r_1$.
5. Open the first row and learn the value $\epsilon_1 := x_1 - r$, and delete the first row. For the bottom sequence $\mathsf{cyc}^{r}(\boldsymbol{z_0})$, apply a rotation rot to ϵ_1-th card from the right, i.e., x_1-th card of $\boldsymbol{z_0}$. Let $\mathsf{cyc}^{r}(\boldsymbol{z_1})$ be the bottom sequence.
6. Apply a composition to each column and make a sequence $\boldsymbol{w_2} = (w_{2,0}, w_{2,1}, \cdots, w_{2,\ell-1})$. (Note that each $w_{2,i}$ contains 2 cards.)

$$\begin{pmatrix} \mathsf{E}_v(-r) \\ \hline \mathsf{cyc}^{r}(\boldsymbol{z_1}) \end{pmatrix} \xrightarrow{\mathsf{Comp}} \boldsymbol{w_2} \xrightarrow{\mathsf{CycShffl}} \mathsf{cyc}^{r_2}(\boldsymbol{w_2}).$$

7. Open the 1st row, and learn the value $\epsilon_2 := -(r + r_2)$. Delete the 1st row, and apply $\mathsf{cyc}^{\epsilon_2}$ to $\mathsf{cyc}^{r+r_2}(\boldsymbol{z_1})$. Output the sequence $\boldsymbol{z_1}$.

$$\begin{pmatrix} \mathsf{E}_v(-(r + r_2)) \\ \hline \mathsf{cyc}^{r+r_2}(\boldsymbol{z_1}) \end{pmatrix} \xrightarrow{\mathsf{Del}} \mathsf{cyc}^{r+r_2}(\boldsymbol{z_1}) \xrightarrow{\mathsf{cyc}^{\epsilon_2}} \boldsymbol{z_1}.$$

Theorem 4. *The above voting protocol is perfectly secure.*

Proof. In the case of $\ell = 3$, the faces of the sequence in the above voting protocol are the following.

$$\begin{pmatrix} \times & \times & \times \\ \times & \times & \times \end{pmatrix} \xrightarrow{\textsf{Insert}} \begin{pmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{pmatrix} \xrightarrow[\textsf{CycShffl}]{\textsf{Comp}} (\times^4, \times^4, \times^4) \xrightarrow{\textsf{Decomp}} \begin{pmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{pmatrix} \xrightarrow{\textsf{Open}}$$

$$\left(\begin{array}{ccc} & \boldsymbol{\epsilon_0} & \\ \hline \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{array}\right) \xrightarrow[\textsf{Rot}]{\textsf{Del}} \begin{pmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{pmatrix} \xrightarrow[\textsf{CycShffl}]{\textsf{Comp}} (\times^3, \times^3, \times^3) \xrightarrow{\textsf{Decomp}} \begin{pmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{pmatrix} \xrightarrow{\textsf{Open}}$$

$$\left(\begin{array}{ccc} & \boldsymbol{\epsilon_1} & \\ \hline \times & \times & \times \\ \times & \times & \times \end{array}\right) \xrightarrow[\textsf{Rot}]{\textsf{Del}} \begin{pmatrix} \times & \times & \times \\ \times & \times & \times \end{pmatrix} \xrightarrow[\textsf{CycShffl}]{\textsf{Comp}} (\times^2, \times^2, \times^2) \xrightarrow{\textsf{Decomp}} \begin{pmatrix} \times & \times & \times \\ \times & \times & \times \end{pmatrix} \xrightarrow{\textsf{Open}}$$

$$\left(\begin{array}{ccc} & \boldsymbol{\epsilon_2} & \\ \hline \times & \times & \times \end{array}\right) \xrightarrow{\textsf{Del}} (\times, \times, \times) \xrightarrow{\textsf{cyc}^{\epsilon_2}} (\times, \times, \times).$$

where $\boldsymbol{\epsilon_i}$ is the face of the opening i.e., $\boldsymbol{\epsilon_i} = (``\epsilon_{i,0}", ``\epsilon_{i,1}", ``\epsilon_{i,2}")$ where $\mathsf{E}_v(\epsilon_i) = (\epsilon_{i,0}, \epsilon_{i,1}, \epsilon_{i,2})$. For any subset $\mathfrak{C} \subsetneq \mathbb{Z}_n$, $\mathcal{S}$ takes $\{x_i\}_{i\in\mathfrak{C}}$ and the face of the output sequence $(\times)$ as inputs, chooses uniformly random $r'_{i,j} \in \mathbb{Z}_m$ and $\epsilon'_{j,k} \in \mathbb{Z}_m$ (for $i \in \mathfrak{C}$, $j \in \{0, 1, \cdots, n\}$, and $k \in \{0, 1, \cdots, \ell - 1\}$), and outputs $(\mathsf{face}(\boldsymbol{C'}), \{x_i, r'_{i,0}, r'_{i,1}, \cdots, r'_{i,n}\}_{i\in\mathfrak{C}})$. (Note that $\mathsf{face}(\boldsymbol{C'})$ can be easily generated from $\epsilon'_{j,k}$ and the transition function π.) This is the same as the real distribution. In general, for an arbitrary ℓ, we can construct the simulator $\mathcal{S}$ similarly. Therefore, our voting protocol is perfectly secure. □

Acknowledgment. The authors would like to thank members of the study group "Shin-Akarui-Angou-Benkyou-Kai" for the valuable discussions and helpful comments, and thank the anonymous reviewers for their comments. This work was partially supported by JSPS KAKENHI Grant Numbers 26330001 and 26330151, Kurata Grant from The Kurata Memorial Hitachi Science and Technology Foundation, and JSPS A3 Foresight Program.

References

1. Crépeau, C., Kilian, J.: Discreet solitary games. In: Stinson, D.R. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 319–330. Springer, Heidelberg (1994)
2. den Boer, B.: More efficient match-making and satisfiability. In: Quisquater, J.-J., Vandewalle, J. (eds.) EUROCRYPT 1989. LNCS, vol. 434, pp. 208–217. Springer, Heidelberg (1990)
3. Mizuki, T., Asiedu, I.K., Sone, H.: Voting with a logarithmic number of cards. In: Mauri, G., Dennunzio, A., Manzoni, L., Porreca, A.E. (eds.) UCNC 2013. LNCS, vol. 7956, pp. 162–173. Springer, Heidelberg (2013)
4. Mizuki, T., Fumishige, U., Sone, H.: Securely computing XOR with 10 cards. Australas. J. Comb. **36**, 279–293 (2006)

5. Mizuki, T., Kumamoto, M., Sone, H.: The five-card trick can be done with four cards. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 598–606. Springer, Heidelberg (2012)
6. Mizuki, T., Shizuya, H.: A formalization of card-based cryptographic protocols via abstract machine. Int. J. Inf. Sec. **13**, 15–23 (2014)
7. Mizuki, T., Shizuya, H.: Practical card-based cryptography. In: Ferro, A., Luccio, F., Widmayer, P. (eds.) FUN 2014. LNCS, vol. 8496, pp. 313–324. Springer, Heidelberg (2014)
8. Mizuki, T., Sone, H.: Six-card secure AND and four-card secure XOR. In: Deng, X., Hopcroft, J.E., Xue, J. (eds.) FAW 2009. LNCS, vol. 5598, pp. 358–369. Springer, Heidelberg (2009)
9. Niemi, V., Renvall, A.: Secure multiparty computations without computers. Theor. Comput. Sci. **191**(1–2), 173–183 (1998)
10. Nishida, T., Hayashi, Y., Mizuki, T., Sone, H.: Card-based protocols for any boolean function. In: Jain, R., Jain, S., Stephan, F. (eds.) TAMC 2015. LNCS, vol. 9076, pp. 110–121. Springer, Heidelberg (2015)
11. Nishida, T., Mizuki, T., Sone, H.: Securely computing the three-input majority function with eight cards. In: Dediu, A.-H., Martín-Vide, C., Truthe, B., Vega-Rodríguez, M.A. (eds.) TPNC 2013. LNCS, vol. 8273, pp. 193–204. Springer, Heidelberg (2013)
12. Shinagawa, K., Mizuki, T., Schuldt, J., Nuida, K., Kanayama, N., Nishide, T., Hanaoka, G., Okamoto, E.: Secure multi-party computation using polarizing cards. In: Tanaka, K., Suga, Y. (eds.) IWSEC 2015. LNCS, vol. 9241, pp. 281–297. Springer, Heidelberg (2015)
13. Stiglic, A.: Computations with a deck of cards. Theor. Comput. Sci. **259**(1–2), 671–678 (2001)