On the Bottleneck Complexity of MPC with Correlated Randomness*

Claudio Orlandi, Divya Ravi, and Peter Scholl

Aarhus University, Denmark
{orlandi,divya,peter.scholl}@cs.au.dk

Abstract. At ICALP 2018, Boyle et al. introduced the notion of the *bottleneck complexity* of a secure multi-party computation (MPC) protocol. This measures the maximum communication complexity of any one party in the protocol, aiming to improve load-balancing among the parties.
In this work, we study the bottleneck complexity of MPC in the preprocessing model, where parties are given correlated randomness ahead of time. We present two constructions of *bottleneck-efficient* MPC protocols, whose bottleneck complexity is independent of the number of parties:

1. A protocol for computing abelian programs, based only on one-way functions.
2. A protocol for selection functions, based on any linearly homomorphic encryption scheme.

Compared with previous bottleneck-efficient constructions, our protocols can be based on a wider range of assumptions, and avoid the use of fully homomorphic encryption.

1 Introduction

Secure Multiparty Computation (MPC) [Yao86,GMW87,BGW88,CCD88] allows a set of mutually distrusting parties to jointly perform a computation on their private inputs in a way no information about their inputs is revealed, except the output of the computation.

There are various fundamental metrics with respect to which the efficiency of an MPC protocol can be measured such as round complexity, communication complexity and computation complexity. Among these, *communication complexity*, which measures the total number of bits communicated by honest parties in the protocol, is often cited as one of the most important ones in practical applications. In this work we study a particular flavour of communication

* Research supported by: the Concordium Blockhain Research Center, Aarhus University, Denmark; the Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM); the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 803096 (SPEC); the Aarhus University Research Foundation (AUFF);the Independent Research Fund Denmark (DFF) under project number 0165-00107B; the Digital Research Centre Denmark (DIREC);

complexity for MPC, namely *bottleneck complexity* ($\mathcal{BC}$). While there has been extensive research aimed at optimizing the communication complexity of MPC protocols, most of these works do not take into account the fact that parties may have asymmetric roles in the protocol, and communication may be unevenly distributed. The work of [BJPY18] addressed this concern by introducing *bottleneck complexity* as a new efficiency metric.

Informally, bottleneck complexity is the maximum communication required by any party within the protocol computation. To illustrate the difference between communication complexity and bottleneck complexity, consider two protocols – say, in the first protocol each party sends a bit to a central party while in the second one, the parties communicate in a chain-like fashion with party P_i sending one bit to P_{i+1} (for $i \in [1, n-1]$, where n is the number of parties). Both these protocols have total communication complexity $\Theta(n)$ but differ significantly in their bottleneck complexities. The first protocol has $\Theta(n)$ bottleneck complexity, while the second has $O(1)$ bottleneck complexity. If the receiving bandwidth of the central party in the first protocol becomes the bottleneck, the second protocol with low bottleneck complexity would be preferred in most practical scenarios. With this motivation, the work of [BJPY18] initiated the study of bottleneck complexity.

In the setting of bottleneck complexity, the focus is on protocols between large number of parties, and the goal is designing protocols with bottleneck complexity *independent of the number of parties*. Such protocols as thus referred as being $\mathcal{BC}$*-efficient*. On the lower bounds side, the work of [BJPY18] shows that, for general functions, achieving even sublinear (in the number of parties n) communication complexity is not always possible – even when no security is required! On the positive side, they present a generic compiler based on fully-homomorphic encryption (FHE) that transforms an insecure MPC protocol into a secure MPC protocol while preserving the bottleneck complexity.

It is well known that homomorphic encryption (in one or other of its many flavours) is a powerful tool for compiling protocols with low communication complexity (see for instance [NN01,IP07,DFH12,AJL+12,LNO13]). However, FHE is still relatively inefficient, and we only know how to construct it using the learning with errors (LWE) assumption [BV11] or the heavy machinery of indistinguishability obfuscation [CLTV15]. It is therefore very natural to ask the question:

For which functions can we achieve low bottleneck complexity without using FHE?

1.1 Our Contribution

In this work, we investigate the feasibility of $\mathcal{BC}$-efficient MPC without using heavy tools such as FHE. Instead, we focus on protocols which make use of correlated randomness and "traditional" assumptions such as one-way functions and (for one of our constructions) linearly homomorphic encryption (which can be instantiated with "90s" style assumptions based on discrete logarithms, factoring,

etc.). All of our protocols are secure against a semi-honest (passive) adversary who may corrupt an arbitrary number of parties.

In this setting, we provide $\mathcal{BC}$-efficient protocols for the following classes of functions:

Abelian Programs. Abelian programs are defined as functions on the sum of the parties' inputs over an abelian group. More formally, an abelian program h takes n elements from an abelian group G as input and outputs $h(x_1, \ldots, x_n) = f(\sum_{i=1}^{n} x_i)$ for some function $f : G \to \{0, 1\}$. This is an expressive class of functions that can be used to securely perform e.g., voting or linear classifiers (see [EOYN21] for more details about applications of abelian programs).

As a warm-up, we design $\mathcal{BC}$-efficient protocols for simple boolean functions such as `AND` and `XOR`, which can be viewed as special cases of abelian programs. These protocols incur bottleneck complexity of $O(\lambda)$ and $O(1)$ respectively. We generalize the approach of these protocols and propose a $\mathcal{BC}$-efficient protocol for abelian programs that has bottleneck complexity $O(\lambda^2)$ (which is independent of n), where λ denotes the security parameter. Our construction is based on garbled circuits, and therefore can be built from one-way functions.

Selection Functions. A selection function is a function of the form $f(x_1 = \mathsf{q}, x_2, \ldots, x_n) = x_{\mathsf{q}}$, where P_1's input is a selection index $\mathsf{q} \in \{2, \ldots, n\}$ and the inputs of the other parties are in $\mathbb{Z}_M$ (set of integers modulo M).

We design a $\mathcal{BC}$-efficient protocol for selection functions that has bottleneck complexity $\mathsf{poly}(\lambda)$ (independent of n), where λ denotes the security parameter. Our construction uses additively homomorphic encryption and garbled circuits as the main tools, which can be instantiated under standard number-theoretic assumptions like decisional Diffie-Hellman, quadratic residuosity, N-th residuosity or learning with errors.

On the communication pattern of $\mathcal{BC}$-efficient protocols. We defer the detailed high-level overviews of the protocols to the respective technical section and highlight an important and common aspect of our $\mathcal{BC}$-efficient constructions below. As a starting point towards designing $\mathcal{BC}$-efficient protocols, we begin by analyzing what types of interaction patterns in MPC support the bottleneck complexity as being independent of the number of parties. The most common interaction pattern in MPC protocols is a complete network (where every pair of parties communicate with each other). Some other popular restricted interaction patterns include 'star' (where all parties interact with a central party) [BGI+14,HIJ+16] and 'chain' (where parties interact over a simple directed path traversing all nodes) [HIJ+16,IMO18]. It is easy to see that the 'chain' interaction pattern is promising to design $\mathcal{BC}$-efficient protocols. This is because it involves each party communicating with only a *constant* number of parties. However, we need two additional properties that a $\mathcal{BC}$-efficient protocol over a chain must satisfy: First, the number of communication traversals or passes over the chain must also be independent of n. Second, the size of the message communicated by each party to its neighbour must also be independent of n. All our protocols thereby entail

a constant number of passes over a chain-like structure, where each message communicated is independent of n. We refer to the technical sections for further details.

Open Problems. As mentioned above, the study of bottleneck complexity in MPC is inherently tied to the bottleneck complexity of protocols *without security*, since not every function can be $\mathcal{BC}$-efficient [BJPY18]. It remains an open question to more thoroughly characterize which functions allow $\mathcal{BC}$-efficient protocols in the clear. With privacy, an interesting challenge is to obtain (even in the correlated randomness model) a compiler that transforms a (possibly insecure) protocol into a secure one with the same bottleneck complexity, while using non-FHE assumptions as considered in this work.

1.2 Related Work

The most relevant work to ours is the work of [BJPY18] which introduced the notion of bottleneck complexity. As mentioned previously, [BJPY18] presents a generic compiler based on fully-homomorphic encryption (FHE) that transforms an insecure MPC protocol into a secure MPC protocol while preserving the bottleneck complexity. For the two-party setting, such a compiler was proposed by the work of [NN01] (this compiler preserved communication complexity; however the notions of bottleneck and communication complexity align in the two-party case). The work of [FKLS20] in the massive parallel computation model focuses on minimizing the storage / communication of servers (which is similar to our goal of minimizing bottleneck complexity). However, similar to [BJPY18], their compiler from an insecure to secure protocol in the parallel computation model is based on FHE (which we wish to avoid).

Related to the setting of MPC with huge number of parties that we consider in this work, the study of scalable MPC was initiated by [DI06] and further explored in works of [DIK$^+$08]. However, these works focus on optimizing communication complexity relative to the circuit size. Similarly, several works on optimizing communication complexity of MPC protocols such as [Cou19,DNPR16,IKM$^+$13] in the information-theoretic setting with correlated randomness and [QWW18,ABJ$^+$19] in the computational setting also focus on regulating the dependence on circuit size. The protocols in these works incur $\Omega(n)$ bottleneck complexity (which is inherent as shown by [BJPY18], since these protocols are for arbitrary functions).

Another related line of work is MPC protocols that involve a one-pass 'chain' interaction pattern, which includes works such as [HIJ$^+$16]. Further, the protocols of [HLP11,GMRW13] that consider consider secure computation in a one-pass client server model can also be adapted to a one-pass chain-based interaction (as pointed out in [HIJ$^+$16]). However, since these works restrict the interaction to a 'single' pass, their constructions achieve residual security (as opposed to standard security). The same holds for efficient non-interactive multiparty computation (NIMPC) constructions in [HIKR18,EOYN21,BGI$^+$14].

Lastly, there are other notions that are related to bottleneck complexity such as communication locality (defined for a party as number of total other parties that the party communicates with) [BGT13] and message complexity (that captures the total number of messages sent in the protocol but does not focus on the size of the message) [IMO18]. However, the proposed protocols optimizing these metrics are for arbitrary functions and incurs $\Omega(n)$ bottleneck complexity.

2 Preliminaries

2.1 Notation

We denote the cryptographic security parameter as λ. We consider a set of $n = n(\lambda)$ parties $\{P_1, \ldots, P_n\}$, where n is polynomially-bounded. The parties are connected by pair-wise secure and authentic channels, and each party is modelled as a probabilistic polynomial time Turing (PPT) machine. We assume that there exists a PPT adversary $\mathcal{A}$, who can *passively* corrupt upto $n-1$ parties. The set of elements $\{1, \ldots, k\}$ is denoted as $[k]$.

2.2 Security Model

We prove the security of our protocols based on the standard real/ideal world paradigm. A reader who is familiar with this may skip to Section 2.3. Essentially, the security of a protocol is analyzed by comparing what an adversary can do in the real execution of the protocol to what it can do in an ideal execution, that is considered secure by definition (in the presence of an incorruptible trusted party). In an ideal execution, each party sends its input to the trusted party over a perfectly secure channel, the trusted party computes the function based on these inputs and sends to each party its respective output. Informally, a protocol is secure if whatever an adversary can do in the real protocol (where no trusted party exists) can be done in the above described ideal computation. In this work, the adversary is assumed to be *passive* (alternately, referred to as being semi-honest) – the corrupt parties must follow the protocol specifications. However, the adversary attempts to learn private information by observing the view of the passively corrupt parties. We refer to [Can00] for further details regarding the security model.

In more detail, let Π be a protocol and $\mathcal{F}$ be a functionality. Let $\mathcal{I}$ denote the set of parties that are corrupt (of size at most $n-1$). The "ideal" world execution involves parties $\{P_1, \ldots, P_n\}$, an ideal adversary $\mathcal{S}$ who controls the parties in $\mathcal{I}$. The "real" world execution involves the PPT parties $\{P_1, \ldots, P_n\}$, and a real world adversary $\mathcal{A}$ who corrupts the parties in $\mathcal{I}$ passively. The *view* of a party in the real world is defined to be its random tape, together with all messages received during the execution of the protocol. In the ideal world, the simulator $\mathcal{S}$ is given as input nothing but the corrupt parties' inputs sent to the trusted party and the outputs they receive from the trusted party. If $\mathcal{S}$ is able

to 'simulate' the real-world view with just this information, intuitively, security must hold. This is formalized below.

We define the following distributions of random variables.

$\texttt{REAL}_{\Pi}(1^{\lambda}, \mathcal{I}; x_1, \ldots, x_n)$: suppose Π is run with security parameter λ where each party P_i runs the protocol honestly using private input x_i. Let V_i denote the view of party P_i at the end of the protocol execution and let y_i denote the output of P_i. Output $\big(\{V_i\}_{i\in\mathcal{I}}, (y_1, \ldots, y_n)\big)$.
$\texttt{IDEAL}_{\mathcal{F},\mathcal{S}}(1^{\lambda}, \mathcal{I}; x_1, \ldots, x_n)$: Let $(y_1, \ldots, y_n) \leftarrow \mathcal{F}(x_1, \ldots, x_n)$. Output $\big(\mathcal{S}(I, \{x_i, y_i\}_{i\in\mathcal{I}}), (y_1, \ldots, y_n)\big)$

A protocol is secure against passive adversaries if the corrupted parties in the real world have views that are indistinguishable from their views in the ideal world.

Definition 1. *A protocol Π securely realizes $\mathcal{F}$ if there exists a PPT ideal world adversary $\mathcal{S}$, such that for every subset of corrupt parties $\mathcal{I}$ and all inputs $x_1, \ldots, x_n$, the following two distributions are computationally indistinguishable:*

$$\texttt{REAL}_{\Pi}(1^{\lambda}, \mathcal{I}; x_1, \ldots, x_n) \overset{c}{\approx} \texttt{IDEAL}_{\mathcal{F},\mathcal{S}}(1^{\lambda}, \mathcal{I}; x_1, \ldots, x_n)$$

2.3 Definitions

Informally, the bottleneck complexity of a protocol is the maximum communication complexity required by any party in the protocol execution. More formally, we have:

Definition 2 (Bottleneck complexity of a Protocol [BJPY18]). *Let $\mathcal{CC}_i(\Pi)$ denote the expected number of bits sent or received by P_i in an execution of Π, with worst-case inputs. The bottleneck complexity of an n-party protocol Π is defined as $\mathcal{BC}(\Pi) = \max_{i\in[n]}\mathcal{CC}_i(\Pi)$*

We note that while we keep the $\mathcal{BC}$-complexity definition of [BJPY18] for consistency, all our protocols satisfy a stronger notion of worse-case (as opposed to expected) $\mathcal{BC}$-complexity.

Definition 3 (Bottleneck complexity of a Function [BJPY18]). *The bottleneck complexity of an n-input function f is the minimum value of $\mathcal{BC}(\Pi)$ when quantified over all n-party distributed protocols Π which securely evaluate f.*

We say that a protocol Π is $\mathcal{BC}$-*efficient*, if the bottleneck complexity of Π is independent of n. Formally, we require that there exists a polynomial $p(\lambda)$ such that for all $n(\lambda) \in \mathsf{poly}(\lambda)$, it holds that $\mathcal{BC}(\Pi) < p(\lambda)$.

Definition 4 (Abelian Programs). *Let G be an abelian group, $S_1, \ldots, S_n$ be subsets of G, and $\mathcal{H}^G_{S_1,\ldots,S_n}$ be the set of functions $h : S_1 \times \cdots \times S_n \to \{0,1\}$ of the form $h(x_1, \ldots, x_n) = f(\Sigma_{i=1}^n x_i)$, for some $f : G \to \{0,1\}$. We call such functions h abelian programs.*

Note that the simple boolean functions of AND and XOR are abelian programs, considering the abelian group G as $\mathbb{Z}_{n+1}$ (integers modulo $n+1$) and the subsets S_i ($i \in [n]$) as the set $\{0,1\}$. $\mathtt{AND}(x_1,\ldots,x_n)$ can be expressed as $f(\sum_{i=1}^{n} x_i)$ where the addition is done modulo $(n+1)$ and $f(x)$ outputs 1 only when $x = n$ and 0 otherwise. On the other hand, $\mathtt{XOR}(x_1,\ldots,x_n)$ can be expressed as $f(\sum_{i=1}^{n} x_i)$ where $f(x)$ outputs $x \mod 2$.

2.4 Primitives

Garbling Scheme. A garbling scheme, introduced by Yao [Yao82] and formalized by Bellare *et al.* [BHR12], enables a party to "encrypt" or "garble" a circuit in such a way that it can be evaluated on inputs — given tokens or "labels" corresponding to those inputs — without revealing what the inputs are.

Definition 5 (Garbling Scheme). *A projective garbling scheme is a tuple of efficient algorithms* $\mathtt{GC} = (\mathtt{garble}, \mathtt{eval})$ *defined as follows.*

$\mathtt{garble}(1^\lambda, \mathtt{C}) \to (\mathtt{GC}, \mathbf{K})$**:** *The garbling algorithm* garble *takes as input the security parameter* λ *and a boolean circuit* $\mathtt{C} : \{0,1\}^\ell \to \{0,1\}^m$*, and outputs a garbled circuit* GC *and* ℓ *pairs of garbled labels* $\mathbf{K} = (K_1^0, K_1^1, \ldots, K_\ell^0, K_\ell^1)$*. For simplicity we assume that for every* $i \in [\ell]$ *and* $b \in \{0,1\}$ *it holds that* $K_\ell^b \in \{0,1\}^\lambda$*.*
$\mathtt{eval}(\mathtt{GC}, K_1, \ldots, K_\ell) \to y$**:** *The evaluation algorithm* eval *takes as input the garbled circuit* GC *and* ℓ *garbled labels* $K_1, \ldots, K_\ell$*, and outputs a value* $y \in \{0,1\}^m$*.*

We require the following properties of a projective garbling scheme:

Correctness. We say GC satisfies *correctness* if for any boolean circuit $\mathtt{C} : \{0,1\}^\ell \to \{0,1\}^m$ and $x = (x_1, \ldots, x_\ell)$ it holds that

$$\Pr[\mathtt{eval}(\mathtt{GC}, \mathbf{K}[x]) \neq \mathtt{C}(x)] = negl(\lambda),$$

where $(\mathtt{GC}, \mathbf{K}) \leftarrow \mathtt{garble}(1^\lambda, \mathtt{C})$ with $\mathbf{K} = (K_1^0, K_1^1, \ldots, K_\ell^0, K_\ell^1)$, and $\mathbf{K}[x] = (K_1^{x_1}, \ldots, K_\ell^{x_\ell})$.

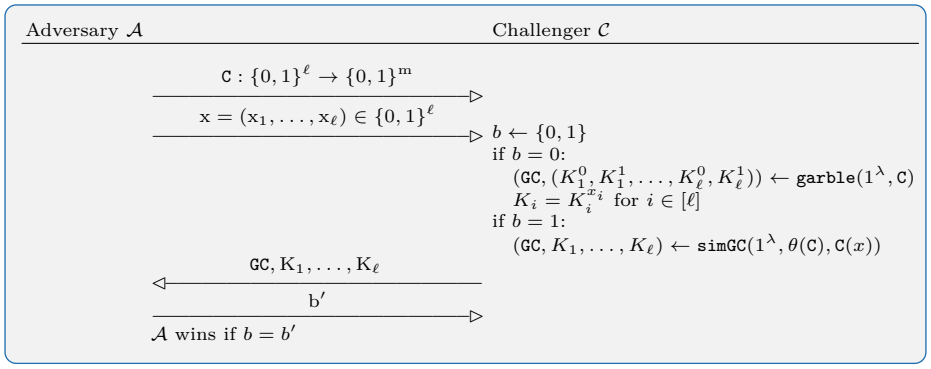
Next, we formally define the security notions we require for a garbling scheme. When garbled circuits are used in such a way that decoding information is used separately, *obliviousness* requires that a garbled circuit together with a set of labels reveals nothing about the input the labels correspond to, and *privacy* requires that the additional knowledge of the decoding information reveals only the appropriate output. In our work, we do not consider decoding information separately (but rather, consider it to be included in the garbled circuit), so we do not need obliviousness.

Privacy. Informally, privacy requires that a garbled circuit together with a set of labels reveal nothing about the input the labels correspond to (beyond the appropriate output and the side-information). For our constructions, we assume the side-information to be the topology of the circuit, denoted as $\theta(C)$.

More formally, we say that $\mathtt{GC}$ satisfies *privacy* if there exists a simulator $\mathtt{simGC}$ such that for every PPT adversary $\mathcal{A}$, it holds that

$$\Pr[\mathcal{A} \text{ wins}] \leq \frac{1}{2} + negl(\lambda)$$

in the following experiment:



Lastly, we remark that one of our constructions requires the use of a slightly modified $\mathtt{garble}$ algorithm that takes as additional input, the labels of the garbled circuit. The modified syntax is as follows: $\mathtt{garble}(1^{\lambda}, \mathtt{C}, \mathbf{K} = (K_1^0, K_1^1, \ldots, K_{\ell}^0, K_{\ell}^1)) \to \mathtt{GC}$. Accordingly, the simulator of the garbling scheme $\mathtt{simGC}$ also takes as input one set of labels i.e. the syntax changes to $\mathtt{simGC}(1^{\lambda}, \theta(\mathtt{C}), \mathtt{C}(x), \{K_1, \ldots, K_{\ell}\}) \to \mathtt{GC}$. Note that most garbled circuits constructions, including Yao's original construction, can be used in this way.

Additively Homomorphic Encryption. We consider linearly homomorphic encryption over $(\mathbb{Z}_M, +)$, the ring of integers modulo M.

Definition 6 (Additively Homomorphic Encryption.). *Let $(\mathbb{Z}_M, +)$ be the ring of integers modulo M. An additively homomorphic encryption scheme over $\mathbb{Z}_M$ is a tuple $\mathtt{AHE} = (\mathtt{Keygen}, \mathtt{Enc}, \mathtt{Dec}, \mathtt{Add}, \mathtt{ScalMul})$ defined as:*

> **Key Generation.** *The algorithm $\mathtt{Keygen}$ is a randomized algorithm that takes as input the security parameter and outputs a public key $\mathtt{pk}$ and a secret key pair $\mathtt{sk}$: $(\mathtt{pk}, \mathtt{sk}) \leftarrow \mathtt{Keygen}(1^{\lambda})$.*
> **Encryption.** *The randomized algorithm $\mathtt{Enc}$ takes as input the public key $\mathtt{pk}$ and the message $m \in \mathbb{Z}_M$ and outputs a ciphertext c : $c \leftarrow \mathtt{Enc}(\mathtt{pk}, m; r)$ (where r denotes the randomness used for encryption).*
> **Decryption.** *The algorithm $\mathtt{Dec}$ takes as input the secret key $\mathtt{sk}$ and the ciphertext c and outputs a plaintext $m \in \mathbb{Z}_M$ (or $\perp$ if the ciphertext is invalid) : $m \leftarrow \mathtt{Dec}(\mathtt{sk}, c)$.*

Homomorphic Addition. *The algorithm* $\texttt{Add}$ *takes as input the public key* $\texttt{pk}$ *and two ciphertexts* c_1 *and* c_2 *and outputs a ciphertext* c^**:* $c^* \leftarrow \texttt{Add}(\texttt{pk}, c_1, c_2)$.
Scalar Multiplication. *The algorithm* $\texttt{ScalMul}$ *takes as input the public key* $\texttt{pk}$*, a ciphertext* c *and an integer* $\alpha \in \mathbb{Z}_M$*, and outputs a ciphertext* $c' : c' \leftarrow \texttt{ScalMul}(\texttt{pk}, c, \alpha)$.

We require the following properties of an $\texttt{AHE}$*:*

Correctness. *An* $\texttt{AHE}$ *is correct if for any* $m \in \mathbb{Z}_M$*,*

$$\Pr\left[\texttt{Dec}(\texttt{sk}, c) \neq m : \begin{array}{c}(\texttt{pk}, \texttt{sk}) \leftarrow \texttt{Keygen}(1^\lambda); \\ c \leftarrow \texttt{Enc}(\texttt{pk}, m)\end{array}\right] \leq negl(\lambda)$$

(where the randomness is taken over the random coins of the algorithms)
Additive Homomorphism. *An* $\texttt{AHE}$ *satisfies additive homomorphism if for any* $m_1, m_2 \in \mathbb{Z}_M$*, the following holds:*

$$\Pr\left[\texttt{Dec}(\texttt{sk}, \texttt{Add}(\texttt{pk}, c_1, c_2)) \neq m_1 + m_2 \bmod M : \begin{array}{c}(\texttt{pk}, \texttt{sk}) \leftarrow \texttt{Keygen}(1^\lambda); \\ c_1 \leftarrow \texttt{Enc}(\texttt{pk}, m_1); \\ c_2 \leftarrow \texttt{Enc}(\texttt{pk}, m_2)\end{array}\right] \leq negl(\lambda)$$

$$\Pr\left[\texttt{Dec}(\texttt{sk}, \texttt{ScalMul}(\texttt{pk}, c, m_2)) \neq m_1 \cdot m_2 \bmod M : \begin{array}{c}(\texttt{pk}, \texttt{sk}) \leftarrow \texttt{Keygen}(1^\lambda); \\ c \leftarrow \texttt{Enc}(\texttt{pk}, m_1)\end{array}\right] \leq negl(\lambda)$$

(where the randomness is taken over the random coins of the algorithms)

CPA Security. *An* $\texttt{AHE}$ *satisfies CPA security if for all PPT adversaries* $\mathcal{A}$*, for* $(\texttt{msg}_0, \texttt{msg}_1) \leftarrow \mathcal{A}(1^\lambda)$*, if* $|\texttt{msg}_0| = |\texttt{msg}_1|$*,*

$$\Pr\left[\mathcal{A}(\texttt{pk}, c) = b : \begin{array}{c}(\texttt{pk}, \texttt{sk}) \leftarrow \texttt{Keygen}(1^\lambda); b \leftarrow \{0,1\}; \\ c \leftarrow \texttt{Enc}(\texttt{pk}, \texttt{msg}_b)\end{array}\right] \leq \frac{1}{2} + negl(\lambda)$$

(where the randomness is taken over the internal coin tosses of $\mathcal{A}$*,* $\texttt{Keygen}$ *and* $\texttt{Enc}$*).*
Circuit Privacy. *An* $\texttt{AHE}$ *satisfies circuit privacy if there exists a simulator* $\mathcal{S}$ *such that for any* $m_1, m_2 \in \mathbb{Z}_M$ *the distributions*

$$\{\texttt{sk}, \mathcal{S}(\texttt{pk}, m_1 + m_2 \bmod M)\} \text{ and } \{\texttt{sk}, \texttt{Add}(\texttt{pk}, \texttt{Enc}(\texttt{pk}, m_1), \texttt{Enc}(\texttt{pk}, m_2))\}$$

$$\{\texttt{sk}, \mathcal{S}(\texttt{pk}, m_1 \cdot m_2 \bmod M)\} \text{ and } \{\texttt{sk}, \texttt{ScalMul}(\texttt{pk}, \texttt{Enc}(\texttt{pk}, m_1), m_2)\}$$

where $(\texttt{pk}, \texttt{sk}) \leftarrow \texttt{Keygen}(1^\lambda)$ *are computationally indistinguishable.*

Note that our definition of circuit privacy implies that $\texttt{Add}$ and $\texttt{ScalMul}$ re-randomize the output ciphertext.

Additively-homomorphic encryption satisfying our requirements can be realized from a variety of assumptions, including QR, DDH, Paillier, learning with errors etc. In the case of DDH, we actually obtain AHE for small integer plaintexts, rather than $\mathbb{Z}_M$. However, this is enough for our application, since our

construction never relies on wraparound modulo M, and we can always guarantee that messages are small by decomposing inputs into blocks and packing into several ciphertexts. Finally, note that standard LWE-based AHE constructions [Reg05] do not support an unbounded number of homomorphic operations; however, in our application this is limited to $O(n)$, so parameters can be chosen accordingly.

3 $\mathcal{BC}$-efficient MPC for Abelian Programs

In this section, we present a $\mathcal{BC}$-efficient MPC protocol for abelian programs. As a warm-up, we begin with describing $\mathcal{BC}$-efficient protocols for basic boolean functions.

3.1 Protocol for AND

At a high-level, the $\mathcal{BC}$-efficient protocol for AND proceeds as follows. The setup maps the potential 1-input of each party to a random group element and distributes it to the respective party. In the online phase, the parties use either the element received as part of the setup or a random element (depending on whether their input is 1 or 0 respectively) to compute the sum incrementally over a chain. The basic idea is that if all parties' inputs are 1, the sum of these group elements would be a special element, say Y, which can also be given as part of the setup and thereby used to determine the output.

Unfortunately, the above protocol idea is susceptible to the residual function attack [1]– suppose the interaction over the chain occurs from P_1 to P_n and imagine the adversary corrupts a subset of parties towards the end of the chain, say P_k to P_n (where k could be any index between 2 to n). In such a case, it is easy to see that the adversary can always learn whether the logical AND of the honest parties' inputs is 1 or not, irrespective of the corrupt parties' inputs. This violates security because as per the ideal functionality, the adversary must not learn this information if any of the corrupt parties' input is 0.

To counter this residual attack, the online phase of our protocol performs an additional backwards pass over the chain i.e. from P_n to P_1. Suppose Y' denotes the sum computed at the end of the forward pass. The backward pass involves n applications of a PRF in a nested manner starting from Y', once by each party using its own PRF key (given as a part of the setup). Intuitively, this prevents the residual attack by the adversary for the following reason – when an honest party computes the PRF using its secret PRF key, the value Y' gets "fixed". It is no longer possible for the adversary to locally compute the output based on the scenario when $Y'' \neq Y'$ was computed at the end of the forward pass of the chain (as he does not know the secret PRF key of the honest party).

[1] However, this approach suffices for residual security, as shown in the NIMPC protocol of [HIKR18].

Lastly, our protocol does another forward pass over the chain to distribute the output. The formal description of the $\mathcal{BC}$-efficient protocol for AND appears in Figure 3.1.

Figure 3.1: $\Pi_{\texttt{AND}}$

Private input. Each party P_i has input bit $x_i \in \{0,1\}$.
Output. $y = x_1 \wedge x_2 \wedge \cdots \wedge x_n$.
Tools. Psuedorandom Function (PRF) $\mathsf{F} : \mathcal{K} \times \{0,1\}^\lambda \to \{0,1\}^\lambda$, where $\mathcal{K}$ refers to the key space
Correlated Randomness Setup. The setup involves the following:

1. For each $i \in [n]$, a uniformly random element $\mathsf{r}_i^1 \leftarrow G$ and a PRF key $\mathsf{s}_i \in \mathcal{K}$ is sampled.
2. Let $Y \leftarrow \Sigma_{i=1}^n \mathsf{r}_i^1$ and $Z \leftarrow \mathsf{F}\Big(\mathsf{s}_1, \mathsf{F}\big(\mathsf{s}_2, \mathsf{F}(\mathsf{s}_3, \ldots \mathsf{F}(\mathsf{s}_{n-2}, \mathsf{F}(\mathsf{s}_{n-1}, \mathsf{F}(\mathsf{s}_n, Y)))\ldots)\big)\Big)$.
3. Output $(Z, \mathsf{r}_i^1, \mathsf{s}_i)$ to each P_i ($i \in [n]$).

The Protocol. The following steps are run in the online phase of the protocol –
Phase 1 (Round 1 to Round n). Each P_i chooses r_i^0 at random at the beginning of Phase 1. During round i, P_i does the following:
– If $i = 1$, set $Y_i = \mathsf{r}_i^{x_i}$.
– If $i \neq 1$, let Y_{i-1} denote the message received from P_{i-1} during the previous round. Compute $Y_i \leftarrow \mathsf{r}_i^{x_i} + Y_{i-1}$.
– If $i < n$, send Y_i to P_{i+1}. Else set $Y' = Y_n$.

Phase 2 (Round $n+1$ to Round $2n$). Each P_i does the following in sequence, starting from $i = n$ to 1:
– If $i = n$, compute $Z_n = \mathsf{F}(\mathsf{s}_n, Y')$.
– If $i \neq n$, receive Z_{i+1} from P_{i+1} in the previous round. Compute $Z_i \leftarrow \mathsf{F}(\mathsf{s}_i, Z_{i+1})$.
– If $i \neq 1$, send Z_i to P_{i-1}. Else set $Z' = Z_1$.

Output Computation. P_1 sets output $y = 1$ if $Z' == Z$ holds; else sets $y = 0$.

Phase 3 (Output Transfer). For i starting from 1 to n, each P_i does the following in sequence -
- If $i \neq 1$, let y denote the output received from P_{i-1} in previous round.
- If $i \neq n$, send y to P_{i+1}.
- Output y.

Correctness. For correctness, note that when $x_i = 1$ for each $i \in [n]$, $Y' = \sum_{i=1}^{n} \mathsf{r}_i^1 = Y$. Therefore, $Z' == Z$ holds and the output y evaluates to 1 in such a case (which is the correct output). Next we consider the case when the input of at least one party is 0 : $Y' \neq Y$ would hold (except with negligible probability) and consequently $Z' \neq Z$ holds (except with negligible probability). The parties output 0 (which is the correct output) in such a case with overwhelming probability.

Security. To argue security, consider the case when the adversary corrupts $n-1$ parties passively and P_i denotes the only honest party. Further, assume that at least one of the corrupt parties' input is 0 (which is the non-trivial case where adversary should learn nothing about x_i; else the input of P_i can be derived from the output itself). Firstly, the adversary learns nothing about x_i from Y_i, as both r_i^0 and r_i^1 are random elements in G. Next, we claim that the adversary learns no information about $Z_i'' = \mathsf{F}\big(\mathsf{s}_i, \mathsf{F}(\mathsf{s}_{i+1}, \mathsf{F}(\mathsf{s}_{i+2}, \ldots, \mathsf{F}(\mathsf{s}_n, Y'')\ldots))\big)$ computed on $Y'' \neq Y'$. This follows from security of the $\mathtt{PRF}$ – In more detail, such an adversary could use the query $Z_{i+1}'' = \mathsf{F}(\mathsf{s}_{i+1}, \mathsf{F}(\mathsf{s}_{i+2}, \ldots, \mathsf{F}(\mathsf{s}_n, Y'')\ldots))$ to distinguish between the $\mathtt{PRF}$ F (where the key sampled is s_i) and a truly random function; thereby breaking the security of the $\mathtt{PRF}$. The above claim ensures that for any $Y'' \neq Y'$, the adversary cannot deduce whether $Z'' \leftarrow \mathsf{F}\big(\mathsf{s}_1, \mathsf{F}(\mathsf{s}_2, \ldots \mathsf{F}(\mathsf{s}_{n-2}, \mathsf{F}(\mathsf{s}_{n-1}, \mathsf{F}(\mathsf{s}_n, Y''))\ldots)\big)$ computed at the end of Phase 2 with respect to Y'' would be identical to Z or not. We can thus conclude that the adversary learns nothing beyond the output 0 and the privacy of P_i's input is maintained. This completes the informal security argument.

$\mathcal{BC}$-Analysis. We analyze the communication incurred by a party, say P_i, in an execution of $\Pi_{\mathtt{AND}}$. First, we observe that throughout $\Pi_{\mathtt{AND}}$, each party communicates with at most two other parties (i.e. P_{i-1} and P_{i+1}). Further, the messages communicated (such as Y_i, Z_i, y) are of at most λ bits. It is therefore easy to see that the bottleneck complexity of $\Pi_{\mathtt{AND}}$ is $O(\lambda)$.

Extension to the dual case of $\mathtt{OR}$ *function.* The above protocol extends naturally to the dual case of the $\mathtt{OR}$ function. For computation of the $\mathtt{OR}$ of all parties' input bits, the setup distributes random elements mapped to potential 0-inputs (instead of 1-inputs as in $\Pi_{\mathtt{AND}}$) and computes Y, Z accordingly. The parties use the information received from the setup in case their input is 0 and sample a random group element otherwise. The rest of the protocol remains the same. It is easy to check that this yields a $\mathcal{BC}$-efficient protocol for $\mathtt{OR}$.

3.2 Protocol for $\mathtt{XOR}$

We present the $\mathcal{BC}$-efficient protocol $\Pi_{\mathtt{XOR}}$ for $\mathtt{XOR}$ in Figure 3.2.

At a high-level, during $\Pi_{\mathtt{XOR}}$, a correlated sharing of 0 is distributed as part of the setup. The parties mask their inputs using their respective share (received as part of the setup). In the online phase, the parties compute the $\mathtt{XOR}$ of their

masked inputs in an incremental manner over a chain. It is easy to see that the XOR of the masked inputs evaluates to the XOR of the parties' inputs.

Note that, unlike the case of AND, $\Pi_{\texttt{XOR}}$ does not need another pass over the chain before output distribution. This is because, in the case of the XOR function, standard security is the same as residual security. Lastly, we point that the correlated sharing of 0 and addition of masked inputs is also used in the NIMPC protocol for addition in [HIKR18]. While their protocol involves parties sending masked inputs to a central party, we carry out the computation in an incremental manner over a chain to preserve the $\mathcal{BC}$-efficiency.

Figure 3.2: $\Pi_{\texttt{XOR}}$

Private input. Each party P_i has input bit $x_i \in \{0,1\}$.
Output. $y = x_1 \oplus x_2 \oplus \cdots \oplus x_n$.
Correlated Randomness Setup. The setup involves the following:

1. For each $i \in [n]$, sample a uniformly random bit $\mathsf{m}_i \leftarrow \{0,1\}$ such that $\oplus_{i=1}^{n} \mathsf{m}_i = 0$.
2. Output m_i to each P_i ($i \in [n]$).

The Protocol. The following steps are run in the online phase of the protocol:

Phase 1 (Round 1 to Round n). At the beginning of Phase 1, each P_i computes $M_i = x_i \oplus \mathsf{m}_i$. During round i, P_i does the following:

- If $i = 1$, set $X_i = M_i$.
- If $i \neq 1$, let X_{i-1} denote the message received from P_{i-1} during the previous round. Compute $X_i \leftarrow X_{i-1} \oplus M_i$.
- If $i < n$, send X_i to P_{i+1}.

Output Computation. P_n computes output y as $y = X_n$.

Phase 2 (Output Transfer). For i starting from n to 1, each P_i does the following in sequence -

- If $i \neq n$, let y denote the output received from P_{i+1} in previous round.
- If $i \neq 1$, send y to P_{i-1}.
- Output y.

Correctness. For correctness, note that $X_n = \oplus_{i=1}^{n} M_i = \oplus_{i=1}^{n}(x_i \oplus \mathsf{m}_i) = \left(\oplus_{i=1}^{n} x_i\right) \oplus \left(\oplus_{i=1}^{n} \mathsf{m}_i\right) = \left(\oplus_{i=1}^{n} x_i\right) \oplus 0 = \oplus_{i=1}^{n} x_i$.

Security. For security, consider the case where there are at least two honest parties, say P_i and P_j (the case of single honest party is trivial as the party's input can be deduced from the output of XOR). During the protocol, the adversary learns $M_i = x_i \oplus \mathsf{m}_i$ and $M_j = x_j \oplus \mathsf{m}_j$. Further since $\oplus_{k=1}^{n} \mathsf{m}_i = 0$, the adversary can deduce $\mathsf{m}_i \oplus \mathsf{m}_j$. However, this leaks nothing beyond $x_i \oplus x_j$ which is allowed as per ideal realization of XOR. This completes the informal security argument.

$\mathcal{BC}$-analysis. We analyze the communication incurred by a party, say P_i, in an execution of $\Pi_{\texttt{XOR}}$. First, we observe that throughout $\Pi_{\texttt{XOR}}$ each party communicates with at most two other parties (i.e. P_{i-1} and P_{i+1}). Further, the messages communicated (such as X_i, y) are a single bit. It is therefore easy to see that the bottleneck complexity of $\Pi_{\texttt{XOR}}$ is $O(1)$.

3.3 Protocol for Abelian Programs

Recall that an abelian program h can be expressed as $h(x_1, \ldots, x_n) = f(\Sigma_{i=1}^{n} x_i)$, for some $f : G \to \{0, 1\}$, where G denotes an abelian group (Definition 4). Towards securely computing such a function in a $\mathcal{BC}$-efficient manner, we begin with the following observation about the protocol design in Section 3.1 and Section 3.2 – At the end of the first forward pass, the parties obtain useful information related to the sum of their inputs. Roughly speaking, this is subsequently used to derive the output by applying some 'special' function. In the case of `AND`, this 'special' function essentially corresponds to checking if the sum is identical to a fixed value that is given to the parties in advance; in the case of `XOR`, the 'special' function turns out to be just the identity function (as the sum computed at the end of the first pass directly yields the output).

Extending this approach to an abelian program $h(x_1, \ldots, x_n) = f(\Sigma_{i=1}^{n} x_i) = f(Y)$, we first note that the useful information related to the sum of parties' inputs (that is computed at the end of the first forward pass) must be such that it does not allow the adversary to learn the sum of the parties' inputs i.e. Y. This is because the output of h may not necessarily reveal Y. For this reason, the first forward pass in our protocol computes a 'masked' sum of inputs, say $Z = Y + R$, where R corresponds to a random mask that is unknown to the parties. Now, the output of h can be derived from this masked sum by computing the 'special' function $f(Z - R)$.

To realize the above computation in a $\mathcal{BC}$-efficient manner, we use garbled circuits. The setup phase involves garbling the circuit corresponding to the 'special' function. In more detail, this circuit takes as input Z, has the mask R hard-coded and computes $f(Z - R)$. Further, to enable the parties to obtain labels for input wires corresponding to Z, the setup additively shares all the labels of the garbled circuit among the parties. Additive sharing of the input labels offers a two-fold advantage – Firstly, it ensures that during the execution of the protocol, the adversary learns at most one label per input wire. This counters residual attacks by the adversary and maintains privacy of honest parties' inputs. Specifically, privacy of garbling guarantees that the adversary is unable to learn the mask R (and therefore Y). Secondly, additive secret sharing supports reconstruction in an incremental fashion. This allows the parties to carry out the reconstruction of the appropriate label corresponding to Z while maintaining the $\mathcal{BC}$-efficiency.

Based on the above high-level ideas, our $\mathcal{BC}$-efficient construction $\Pi_{\texttt{abl}}$ has the following structure. It begins with a forward pass to compute the masked sum Z. Next, it does a backward pass where the parties use their respective additive shares (received as part of the setup) and reconstruct the labels corresponding to Z. Using these labels, the garbled circuit (received as a part of the setup)

computing $f(Z - R)$ can be evaluated, which results in the output of h. The formal description of $\Pi_{\texttt{abl}}$ appears in Figure 3.3.

Figure 3.3: $\Pi_{\texttt{abl}}$

Private input. Each party P_i has input $x_i \in G$, where G denotes an abelian group.
Output. $y = h(x_1, \ldots, x_n) = f(\Sigma_{i=1}^{n} x_i)$.
Tools. Garbling scheme $(\texttt{garble}, \texttt{eval})$
Correlated Randomness Setup. The setup involves the following:

1. For each $i \in [n]$, sample $\mathsf{r}_i \in G$. Let $\Sigma_{i=1}^{n} \mathsf{r}_i = R$
2. Let $C_R(Z)$ denote a circuit that has $R \in \{0,1\}^\lambda$ [a] hard-coded, takes as input $Z \in \{0,1\}^\lambda$ and outputs $f(Z - R)$. Compute $(\texttt{GC}, \{K_\alpha^{(0)}, K_\alpha^{(1)}\}_{\alpha \in [\lambda]}) \leftarrow \texttt{garble}(C, 1^\lambda)$ [b].
3. For each $\alpha \in [\lambda], b \in \{0,1\}$ – let $(K_{\alpha,1}^{(b)}, \ldots, K_{\alpha,n}^{(b)})$ denote the additive sharing of $K_\alpha^{(b)}$ i.e. $\Sigma_{i=1}^{n} K_{\alpha,i}^{(b)} = K_\alpha^{(b)}$.
4. Output $(\texttt{GC}, \mathsf{r}_i, \{K_{\alpha,i}^{(0)}, K_{\alpha,i}^{(1)}\}_{\alpha \in [\lambda]})$ to P_i for each $i \in [n]$.

The Protocol. The following steps are run in the online phase of the protocol:
Phase 1 (Round 1 to Round n). Each P_i sets $M_i := x_i + \mathsf{r}_i$. During round i, P_i does the following:

- If $i = 1$, let $Y_i = M_i$.
- If $i \neq 1$, let Y_{i-1} denote the message received from P_{i-1} during the previous round. Compute $Y_i \leftarrow Y_{i-1} + M_i$.
- If $i < n$, send Y_i to P_{i+1}. Else, set $Z = Y_n$.

Phase 2 (Round $n+1$ to Round $2n$). Each P_i does the following in sequence, starting from $i = n$ to 1:

- If $i = n$, let $(z_1, z_2, \ldots, z_\lambda)$ denote the bits corresponding to Z. For each $\alpha \in [\lambda]$, set $K'_{\alpha,n} = K_{\alpha,n}^{(z_\alpha)}$.
- If $i \neq n$, parse the message received from P_{i+1} in the previous round as $\left(Z = (z_1, z_2, \ldots, z_\lambda), \{K'_{\alpha,i+1}\}_{\alpha \in [\lambda]}\right)$.
- For each $\alpha \in [\lambda]$, compute $K'_{\alpha,i} \leftarrow K'_{\alpha,i+1} + K_{\alpha,i}^{(z_\alpha)}$.
- If $i \neq 1$, send $\left(Z, \{K'_{\alpha,i}\}_{\alpha \in [\lambda]}\right)$ to P_{i-1}. Else, set $\vec{K} = \{K'_{\alpha,1}\}_{\alpha \in [\lambda]}$.

Output Computation. P_1 runs $y \leftarrow \texttt{eval}(\texttt{GC}, \vec{K})$.

Phase 3 (Output Transfer). This phase is identical to Phase 3 of Figure 3.1.

[a] We assume a canonical mapping from elements in G to strings in $\{0,1\}^\lambda$.
[b] Here, we assume for simplicity that the garbling scheme has the property that the garbled circuit reveals no information about the hard-coded value R. This

property holds for Yao's garbled circuits [Yao86]. However, this requirement can easily be removed by defining the circuit C to take an additional input R (instead of hard-coding R), for which the labels are given directly to the parties as part of the setup.

Correctness. For correctness, note that Z computed by P_n at the end of Phase 1 is $Z = \Sigma_{i=1}^{n} M_i = \Sigma_{i=1}^{n}(x_i + \mathsf{r}_i) = \Sigma_{i=1}^{n} x_i + \Sigma_{i=1}^{n} \mathsf{r}_i = \Sigma_{i=1}^{n} x_i + R$. Thus, the output computed via the evaluation of the garbled circuit as $f(Z - R) = f(\Sigma_{i=1}^{n} x_i) = h(x_1, \ldots, x_n)$ is indeed the correct output.

$\mathcal{BC}$-Analysis. We analyze the communication incurred by a party, say P_i, in an execution of $\Pi_{\mathtt{abl}}$. First, we observe that throughout $\Pi_{\mathtt{abl}}$ each party communicates with at most two other parties (i.e. P_{i-1} and P_{i+1}). Further, the messages communicated (such as $Y_i, Z, \{K'_{\alpha,i}\}_{\alpha\in[\lambda]}, y$) are of size at most λ^2. It is therefore easy to see that the bottleneck complexity of $\Pi_{\mathtt{abl}}$ is $O(\lambda^2)$.

We state the formal theorem below.

Theorem 1. *Protocol $\Pi_{\mathtt{abl}}$ securely computes the abelian program h against an adversary corrupting upto $n-1$ parties passively.*

Proof. Let $\mathcal{I}$ and $\mathcal{H} = \mathcal{P} \setminus \mathcal{I}$ denote the set of indices corresponding to corrupt and honest parties respectively. Let $j_{\min}$ and $j_{\max}$ denote the least and maximum index corresponding to an honest party, where the indices are in the range $\{1, \ldots, n\}$. To prove security, we define below a simulator $\mathcal{S}$ that simulates the real-world view of the parties. Recall that $\mathcal{S}$ is given $(\mathcal{I}, \{x_i\}_{i\in\mathcal{I}}, y)$ (see Section 2.2 for details about the security model).

Setup Simulation. Run $(\mathtt{GC}, \vec{K} = \{K_1, \ldots, K_\lambda\}) \leftarrow \mathtt{simGC}(1^\lambda, \theta(C), y)$, where $\mathtt{simGC}$ denotes the simulator of the garbling scheme and $\theta(C)$ denotes the topology of the circuit computing $f(Z - R)$ (note that the topology is independent of the hard-coded value R). For each $i \in \mathcal{I}$, sample $\left(\mathsf{r}_i, \{K^{(0)}_{\alpha,i}, K^{(1)}_{\alpha,i}\}_{\alpha\in[\lambda]}\right)$ at random. The view of corrupted P_i constructed by $\mathcal{S}$ at this stage comprises of $\left(\mathtt{GC}, \mathsf{r}_i, \{K^{(0)}_{\alpha,i}, K^{(1)}_{\alpha,i}\}_{\alpha\in[\lambda]}\right)$.

Phase 1 Simulation. Choose Y_j at random on behalf of each honest P_j ($j \in \mathcal{H}$). For $i \in \mathcal{I}$ and $j \in \mathcal{H}$, add Y_j to a corrupt party P_i's view if $i = j + 1$.

Phase 2 Simulation. Recall that for each $i \in \mathcal{I}$, $\mathcal{S}$ knows x_i and defined r_i (during the simulation of the setup) and therefore can compute M_i.

- Compute Z as $Z = Y_{j_{\max}} + \sum_{k=j+1}^{n} M_k$. Parse $Z = (z_1, z_2, \ldots, z_\lambda)$.
- For each $\alpha \in [\lambda]$ and $j \in \mathcal{H}$: If $j \neq j_{\min}$, sample $K'_{\alpha,j}$ at random. Else (i.e. for $j = j_{\min}$), set $K'_{\alpha,j_{\min}}$ such that $K'_{\alpha,j_{\min}} + \sum_{i=1}^{j_{\min}-1} K^{(z_\alpha)}_{\alpha,i} = K_\alpha$. Note that $K^{(z_\alpha)}_{\alpha,i}$ for $i \in [j_{\min} - 1]$ corresponds to additive shares of corrupt parties which were already defined by $\mathcal{S}$ during the setup.
- For $i \in \mathcal{I}$ and $j \in \mathcal{H}$, add $(Z, K'_{\alpha,j})$ to a corrupt party P_i's view if $i = j - 1$.

Phase 3 Simulation. For $i \in \mathcal{I}$ and $j \in \mathcal{H}$, add y to a corrupt party P_i's view if $i = j + 1$.

Below, we argue that the views of corrupt parties in the real and ideal world are indistinguishable via a series of intermediate hybrids.

-Hyb_0: Same as the real-world execution.

-Hyb_1: Same as Hyb_0, except that Y_j for $j \in \mathcal{H}$ is sampled at random.

This is in contrast to the previous hybrid where Y_j is computed as $Y_j = Y_{j-1} + x_j + \mathsf{r}_j$. Since r_j is uniformly distributed in G, this is indistinguishable from the previous hybrid.

-Hyb_2: Same as Hyb_1, except for the way in which $K'_{\alpha,j}$ for $j \in \mathcal{H}, \alpha \in [\lambda]$ is computed – For $j \neq j_{\min}$, $K'_{\alpha,j}$ is chosen uniformly at random, whereas for $j = j_{\min}$, it is set such that $K'_{\alpha,j_{\min}} + \sum_{i=1}^{j_{\min}-1} K_{\alpha,i}^{(z_\alpha)} = K_\alpha^{(z_\alpha)}$.

This is in contrast to the previous hybrid where $K'_{\alpha,j}$ is computed as $K'_{\alpha,j} \leftarrow K'_{\alpha,j+1} + K_{\alpha,j}^{(z_\alpha)}$. Since $K_{\alpha,j}^{(z_\alpha)}$ is distributed uniformly at random (conditioned on $\Sigma_{k=1}^{n} K_{\alpha,k}^{(z_\alpha)} = K_\alpha^{(z_\alpha)}$), this is indistinguishable from the previous hybrid.

-Hyb_3: Same as Hyb_2, except that $(\mathtt{GC}, \vec{K})$ is computed as $(\mathtt{GC}, \vec{K}) \leftarrow \mathtt{simGC}(1^\lambda, \theta(C), y)$, where $\theta(C)$ denotes the topology of the circuit computing $f(Z - R)$.

This is in contrast to Hyb_2 where $(\mathtt{GC}, \vec{K})$ is computed as $(\mathtt{GC}, \{K_\alpha^{(z_\alpha)}\}_{\alpha\in[\lambda]})$, where $(\mathtt{GC}, \{K_\alpha^{(0)}, K_\alpha^{(1)}\}_{\alpha\in[\lambda]}) \leftarrow \mathtt{garble}(C, 1^\lambda)$. It follows from privacy of the garbling scheme (see Definition 5) that Hyb_3 is indistinguishable from Hyb_2.

Since Hyb_3 corresponds to the ideal execution and every pair of consecutive hybrids are indistinguishable, this completes the proof that the views of corrupt parties in the real and ideal world are indistinguishable.

4 $\mathcal{BC}$-efficient Protocol for Selection Functions

In this section, we present a $\mathcal{BC}$-efficient protocol $\Pi_{\mathtt{sel}}$ for the selection function $f(x_1 = \mathsf{q}, x_2, \ldots, x_n) = x_\mathsf{q}$, where P_1's input is a selection index $\mathsf{q} \in \{2, \ldots, n\}$ and the inputs of the other parties are in $\mathbb{Z}_M$, the plaintext space of an additively homomorphic encryption scheme.

As a starting point, note that the output of the selection function can be viewed as $\sum_{i=2}^{n}(b_i \cdot x_i)$, where $b_i \in \{0, 1\}$ denotes an indicator bit showing whether $i = \mathsf{q}$ holds or not (i.e. $b_i = 1$ if $i = \mathsf{q}$ and 0 otherwise). This seems promising as such a computation can be carried out in a chain – each party P_i computes $(b_i \cdot x_i)$, and these values can be aggregated while preserving $\mathcal{BC}$-efficiency as seen in Section 3. Unfortunately, this direct approach requires P_i

to know b_i, which must not be allowed (as it is not revealed by output of f). However, if P_i somehow had access to an encryption of b_i instead, then the above computation over the chain could be carried out under the hood of *additive homomorphic encryption*; maintaining that b_i remains private from P_i.

Next, we note that since b_i depends on the input of party P_1, it is not possible to distribute encryptions of b_i directly to P_i during the input-independent setup. Therefore, to account for each possible value of $x_1 = \mathsf{q}$ (where $\mathsf{q} \in [n]$), the setup distributes to each P_i ($i \geq 2$) a look-up table containing n ciphertexts: among these ciphertexts, the one corresponding to $\mathsf{q} = i$ would be an encryption of $b_i = 1$, while the others would correspond to encryptions of $b_i = 0$. The idea is to 'point' P_i to the appropriate ciphertext in the look-up table without revealing b_i. For this, a random cyclic-shift can be used, say using an offset r (unknown to P_i). This offset r is given to P_1 to enable her to compute the 'pointer' $\mathsf{q}' = \mathsf{q} + r$. Lastly, the encryption is assumed to be randomized, otherwise P_i could learn b_i by simply inspecting her look-up table (since all but one ciphertexts correspond to encryptions of 0).

Based on the above ideas, we can obtain a $\mathcal{BC}$-efficient construction for securely computing the selection function, with $O(n)$ storage costs, as follows (in our final construction, we reduce the storage overhead). During the setup, each P_i ($i \geq 2$) receives a look-up table containing a 'shifted' sequence of n ciphertexts (as explained above). Each of the look-up tables uses the same offset r for the shift and P_1 is given this offset r. The online phase begins with P_1 computing the appropriate pointer $\mathsf{q}' = x_1 + r = \mathsf{q} + r$, which is communicated over a forward pass of the chain to all. During this pass, each party P_i uses the ciphertext at index q' of her look-up table (which would correspond to an encryption of b_i) to homomorphically compute an encryption of $(b_i \cdot x_i)$. These encryptions are aggregated over the chain, resulting in the encryption of $\sum_{i=2}^{n}(b_i \cdot x_i) = x_{\mathsf{q}}$ at the end of the forward pass. The final step is to compute the output by decrypting this $\mathtt{AHE}$ output ciphertext. Possible ways to do this decryption in a $\mathcal{BC}$- efficient manner include either **(a)** incremental decryption (carried out over a chain) of this ciphertext or **(b)** use garbled circuits. We opt for approach **(b)** to avoid the additional requirement that the $\mathtt{AHE}$ used must support incremental decryption.

In the approach using garbled circuits for decryption, we consider a garbled circuit that takes as input the $\mathtt{AHE}$ ciphertext, has the secret key of $\mathtt{AHE}$ hard-coded and outputs the decryption. Similar to the protocol $\Pi_{\mathtt{abl}}$ (Figure 3.3), the input labels of the garbled circuit are additively shared among the parties. This enables them to reconstruct (over a backward pass of the chain) the appropriate label corresponding to the output ciphertext and obtain the output via evaluation of the garbled circuit.

While the above construction is indeed $\mathcal{BC}$-efficient in the online phase, observe that the setup involves parties receiving look-up tables of size n. We avoid this in our final construction that achieves storage and computation complexity also independent of n. The main idea is to 'compress' the look-up table in a way that still allows party P_i to obtain the ciphertext corresponding to b_i (without revealing b_i).

Interestingly, this can be done using garbled circuits, thereby avoiding new primitives or assumptions over the above described construction. As part of the setup, each P_i is given a garbled circuit, say $\mathtt{GC}_i$ which garbles a circuit C described as follows: The circuit C has i, the public key of the $\mathtt{AHE}$ and the randomness used for encryption hard-coded; it takes as input the index j and outputs the encryption of b_i (where $b_i = 1$ if $\mathtt{q} = i$ holds and 0 otherwise). Further, P_1 is given the labels for input wires. The final construction differs from the previous construction in the following aspects: **(a)** the 'pointer' sent by P_1 is the appropriate label corresponding to $\mathtt{q}$ (instead of the index $\mathtt{q}'$ in the look-up table approach). **(b)** Each P_i obtains the encryption of b_i by evaluating $\mathtt{GC}_i$ using the label corresponding to $\mathtt{q}$ that it receives in the forward pass (instead of obtaining the appropriate encryption directly from the look-up table).

To make the above approach work while preserving $\mathcal{BC}$-efficiency, it is important that the 'pointer' label given by P_1 can be used by all parties to evaluate their respective garbled circuits (analogous to P_1 giving the same pointer index to all in the lookup-table approach). This is because we cannot afford to make P_1 give n different pointers, one for each garbled circuit as that would inflate the $\mathcal{BC}$ complexity. To enable the use of the same input label across the n garbled circuits, we use a slightly modified garbling algorithm that takes as input the labels corresponding to the input wires. The garbling algorithm of Yao [Yao86] easily supports this.

Lastly, we point out that it may seem problematic to give P_1 all the input labels of the garbled circuits (computing the encryptions) as this would compromise the privacy of garbling. However, we need to rely of privacy of garbling only when P_1 is honest. This is because an adversary corrupting P_1 and P_i ($i \geq 2$) already knows b_i. When P_1 is honest, a potentially corrupt P_i ($i \geq 2$) will have access to only one set of input labels of $\mathtt{GC}_i$ and privacy of garbling ensures that P_i cannot learn b_i. Thus, security is maintained.

The formal description of the protocol appears in Figure 4.1.

Figure 4.1: $\Pi_{\mathtt{sel}}$

Private input. P_1's input is a selection index $\mathtt{q} \in \{2, \ldots, n\}$ and the inputs of the other parties are $x_i \in \mathbb{Z}_M$.
Output. $f(x_1 = \mathtt{q}, x_2, \ldots, x_n)$.
Tools. Additively homomorphic Encryption over $\mathbb{Z}_M$ $\mathtt{AHE} = (\mathtt{Keygen}, \mathtt{Enc}, \mathtt{Dec}, \mathtt{Add}, \mathtt{ScalMul})$ (see Definition 6)[a], Garbling scheme $(\mathtt{garble}, \mathtt{eval})$ [b].
Correlated Randomness Setup. The setup involves the following:

1. Let $(\mathtt{pk}, \mathtt{sk}) \leftarrow \mathtt{Keygen}(1^\lambda)$.
2. For $i \in \{2, \ldots, n\}$:
   - Let $C_{i,\mathtt{pk},r_{i,\mathtt{Enc}}}(\mathtt{q})$ denote a circuit that has the value i, $\mathtt{pk}$ and the randomness used for encryption $r_{i,\mathtt{Enc}}$ hard-coded, takes as input an index $\mathtt{q}$; outputs $c = \mathtt{Enc}(\mathtt{pk}, 1; r_{i,\mathtt{Enc}})$ if $\mathtt{q} = i$ holds and $c = \mathtt{Enc}(\mathtt{pk}, 0; r_{i,\mathtt{Enc}})$ otherwise.

- Sample a set of input labels $\{\widetilde{K}_\alpha^{(0)}, \widetilde{K}_\alpha^{(1)}\}_{\alpha\in[\lambda]}$

– Compute $\mathtt{GC}_i \leftarrow \mathtt{garble}(C_{i,\mathtt{pk},r_{\mathtt{Enc}}}, \{\widetilde{K}_\alpha^{(0)}, \widetilde{K}_\alpha^{(1)}\}_{\alpha\in[\lambda]}, 1^\lambda)$

3. Let $C_{\mathtt{sk}}(ct)$ denote a circuit that has a secret key value $\mathtt{sk}$ hard-coded, takes as input a ciphertext ct and outputs $z \leftarrow \mathtt{Dec}(\mathtt{sk}, ct)$.
4. Sample a set of input labels $\{K_\alpha^{(0)}, K_\alpha^{(1)}\}_{\alpha\in[\lambda]}$ and compute $\mathtt{GC}_{\mathtt{Dec}} \leftarrow \mathtt{garble}(C_{\mathtt{sk}}, \{K_\alpha^{(0)}, K_\alpha^{(1)}\}_{\alpha\in[\lambda]}, 1^\lambda)$. For each $\alpha \in [\lambda], b \in \{0,1\}$, let $(K_{\alpha,1}^{(b)}, \ldots, K_{\alpha,n}^{(b)})$ denote the additive sharing of $K_\alpha^{(b)}$.
5. Output $\left(\mathtt{pk}, \mathtt{GC}_i, \{K_{\alpha,i}^{(0)}, K_{\alpha,i}^{(1)}\}_{\alpha\in[\lambda]}\right)$ to P_i $(i \in \{2, \ldots, n\})$.
6. Output $\left(\mathtt{pk}, \{\widetilde{K}_\alpha^{(0)}, \widetilde{K}_\alpha^{(1)}\}_{\alpha\in[\lambda]}, \{K_{\alpha,1}^{(0)}, K_{\alpha,1}^{(1)}\}_{\alpha\in[\lambda]}, \mathtt{GC}_{\mathtt{Dec}}\right)$ to P_1.

The Protocol. The following steps are run in the online phase of the protocol:

Phase 1 (Round 1 to Round n). During round i, P_i does the following:

– If $i = 1$: let $(b_1, \ldots, b_\lambda)$ denote the bits corresponding to $x_1 = \mathtt{q}$. Sample randomness r and compute $ct_1 = \mathtt{Enc}(\mathtt{pk}, 0; r)$.
Set $\mathbf{K} = \{\widetilde{K}_\alpha^{(b_\alpha)}\}_{\alpha\in[\lambda]}$. Send $(\mathbf{K}, ct_1)$ to P_{i+1}.

– If $i \neq 1$: let $(\mathbf{K}, ct_{i-1})$ denote the message received from P_{i-1} during the previous round.
- Compute $c_i' = \mathtt{eval}(\mathtt{GC}_i, \mathbf{K})$.
- Compute $c_i^* = \mathtt{ScalMul}(\mathtt{pk}, c_i', x_i)$.
- Compute $ct_i = \mathtt{Add}(\mathtt{pk}, ct_{i-1}, c_i^*)$.

– If $i < n$, send $(\mathbf{K}, ct_i)$ to P_{i+1}. Else, set $Z = ct_n$.

Phase 2 (Round $n+1$ to Round $2n$). In this phase, parties use their additive shares of the labels of $\mathtt{GC}_{\mathtt{Dec}}$ to reconstruct the input label (say $\vec{K}$) corresponding to Z. The steps are identical to Phase 2 of Figure 3.3 .

Output Computation. P_1 runs $y \leftarrow \mathtt{eval}(\mathtt{GC}_{\mathtt{Dec}}, \vec{K})$.

Phase 3 (Output Transfer). This phase is identical to Phase 3 of Figure 3.1.

[a] Here we assume for simplicity that $M \geq 2^\lambda$ so the input of the parties can be encrypted using the $\mathtt{AHE}$. As described in the preliminaries, if $M < 2^\lambda$, one can simply perform the encryption of the inputs e.g., in a bit by bit fashion.

[b] In this construction, we assume the modified $\mathtt{garble}$ algorithm which takes as additional input, the labels of the garbled circuit. Yao's garbling supports this requirement easily.

Correctness. Correctness of garbling with respect to $\mathtt{GC}_i$ for each $P_i \in [n]$ and correctness of $\mathtt{AHE}$ ensures that Z computed at the end of the first forward pass corresponds to an encryption of $\sum_{i=2}^n (b_i \cdot x_i) = x_{\mathtt{q}}$ (where $b_i = 0$ for $i \neq \mathtt{q}$ and

1 otherwise). It now follows from correctness of garbling with respect to $\mathtt{GC}_{\mathtt{Dec}}$ (that computes the decryption of an input AHE ciphertext) that the output y computed is indeed the correct output x_{q}.

$\mathcal{BC}$-Analysis. We analyze the communication incurred by a party, say P_i, in an execution of $\Pi_{\mathtt{sel}}$. First, we observe that throughout $\Pi_{\mathtt{sel}}$ each party communicates with at most two other parties (i.e. P_{i-1} and P_{i+1}). Further, the messages communicated (such as $\mathbf{K}, \{K'_{\alpha,i}\}_{\alpha\in[\lambda]}, y$) are of size at most λ^2, for the GC wire label shares, plus the size of one AHE ciphertext, which is $\mathsf{poly}(\lambda)$. It is therefore easy to see that the bottleneck complexity of $\Pi_{\mathtt{sel}}$ is $\mathsf{poly}(\lambda)$ and independent of n.

Extension to larger inputs. The protocol $\Pi_{\mathtt{sel}}$ can be extended for the case where the inputs of $P_2, \ldots P_n$ are arbitary-length vectors i.e. $x_i \in \mathbb{Z}_M^k$, by running the scalar multiplication on each entry of the input vector. In more detail, each P_i is still given a single garbled circuit $\mathtt{GC}_i$ which he uses to compute c'_i. However, each P_i would now compute a set of k ciphertexts $\{c^*_{i,\alpha} \leftarrow \mathtt{ScalMul}(\mathtt{pk}, c'_i, x_{i,\alpha})\}_{\alpha\in[k]}$ and $\{ct_{i,\alpha} = \mathtt{Add}(\mathtt{pk}, ct_{i-1,\alpha}, c^*_{i,\alpha})\}_{\alpha\in[k]}$ accordingly, where $x_{i,\alpha}$ denotes the α'th entry of x_i. This would introduce a multiplicative factor of k (which is independent of n) in the $\mathcal{BC}$ complexity of $\Pi_{\mathtt{sel}}$, thereby maintaining the $\mathcal{BC}$-efficiency.

We state the formal theorem below.

Theorem 2. *Protocol $\Pi_{\mathtt{sel}}$ securely computes the selection function $f(x_1 = j, x_2, \ldots, x_n) = x_j$ (where $x_1 \in \{2, \ldots, n\}$ and $x_i \in \mathbb{Z}_M$, $i \geq 2$) against an adversary corrupting upto $n-1$ parties passively.*

Proof. Let $\mathcal{I}$ and $\mathcal{H} = \mathcal{P} \setminus \mathcal{I}$ denote the set of indices corresponding to corrupt and honest parties respectively. Let $j_{\min}$ and $j_{\max}$ denote the least and maximum index corresponding to an honest party, where the indices are in the range $\{1, \ldots, n\}$. To prove security, we define below a simulator $\mathcal{S}$ that simulates the real-world view of the parties. Recall that $\mathcal{S}$ is given $(\mathcal{I}, \{x_i\}_{i\in\mathcal{I}}, y)$ (we refer to Section 2.2 for details about the security model).

Since the parties' roles are asymmetric, we describe the simulation in two parts based on whether $P_1 \in \mathcal{I}$ or not. First, we describe the simulation for the case when $P_1 \in \mathcal{I}$. As discussed previously, we do not rely on privacy of garbling with respect to $\mathtt{GC}_i$ ($i \in \mathcal{I}$) in this case. However, privacy of garbling with respect to $\mathtt{GC}_{\mathtt{Dec}}$ is crucial.

Setup Simulation.

- Compute the values $\mathtt{pk}$, $\mathtt{GC}_i$, $\{\widetilde{K}^{(0)}_\alpha, \widetilde{K}^{(1)}_\alpha\}_{\alpha\in[\lambda]}$ for each $i \in \mathcal{I}$ as per the protocol steps in Figure 4.1.
- Sample $\vec{K} = \{K_1, \ldots, K_\lambda\}$ and run $\mathtt{GC}_{\mathtt{Dec}} \leftarrow \mathtt{simGC}(1^\lambda, \theta(C), y, \vec{K})$, where simGC denotes the simulator of the garbling scheme and $\theta(C)$ denotes the circuit computing the decryption of an AHE input ciphertext. (Note that the topology of the circuit is independent of the hard-coded values).
- For each $i \in \mathcal{I}$, sample $\left(\{K^{(0)}_{\alpha,i}, K^{(1)}_{\alpha,i}\}_{\alpha\in[\lambda]}\right)$ at random.

The view of corrupted P_i ($i \geq 2$) constructed by $\mathcal{S}$ at this stage comprises of $\left(\mathtt{pk}, \mathtt{GC}_i, \{K^{(0)}_{\alpha,i}, K^{(1)}_{\alpha,i}\}_{\alpha\in[\lambda]}\right)$. The view of corrupted P_1 constructed by $\mathcal{S}$ at this stage comprises of $\left(\mathtt{pk}, \{\widetilde{K}^{(0)}_{\alpha}, \widetilde{K}^{(1)}_{\alpha}\}_{\alpha\in[\lambda]}, \{K^{(0)}_{\alpha,1}, K^{(1)}_{\alpha,1}\}_{\alpha\in[\lambda]}, \mathtt{GC}_{\mathtt{Dec}}\right)$.

Phase 1 Simulation.

- Recall that $\mathcal{S}$ knows $x_1 = (b_1, \ldots, b_\lambda)$ and defined $\{\widetilde{K}^{(0)}_{\alpha}, \widetilde{K}^{(1)}_{\alpha}\}_{\alpha\in[\lambda]}$ during the setup simulation and can therefore compute $\mathbf{K} = \{\widetilde{K}^{(b_\alpha)}_{\alpha}\}_{\alpha\in\lambda}$.
- On behalf of each honest P_j ($j \in \mathcal{H}$), compute $ct_j \leftarrow \mathcal{S}_{\mathtt{AHE}}(\mathtt{pk}, m)$, where $m = y$ if $x_1 = \mathsf{q} \leq j$ and $m = 0$ otherwise. Here, $\mathcal{S}_{\mathtt{AHE}}$ refers to the simulator of the $\mathtt{AHE}$ for circuit privacy (Definition 6).

For $i \in \mathcal{I}$ and $j \in \mathcal{H}$, add $(\mathbf{K}, ct_j)$ to a corrupt party P_i's view if $i = j + 1$.

Phase 2 Simulation. This is similar to Phase 2 simulation of Thm 1, which we describe below for completeness. Recall that for each $i \in \mathcal{I}$, $\mathcal{S}$ knows x_i and distributed $\mathtt{GC}_i$ (during the simulation of the setup) and can therefore can compute c^*_i.

- Compute Z by homomorphic addition of $ct_{j_{\max}}$ and each c^*_k for $k \in [j+1, n]$. Parse $Z = (z_1, z_2, \ldots, z_\lambda)$.
- For each $\alpha \in [\lambda]$ and $j \in \mathcal{H}$: If $j \neq j_{\min}$, sample $K'_{\alpha,j}$ at random. Else (i.e. for $j = j_{\min}$), set $K'_{\alpha,j_{\min}}$ such that $K'_{\alpha,j_{\min}} + \sum_{i=1}^{j_{\min}-1} K^{(z_\alpha)}_{\alpha,i} = K_\alpha$. Note that $K^{(z_\alpha)}_{\alpha,i}$ for $i \in [j_{\min} - 1]$ corresponds to additive shares of corrupt parties which were already defined by $\mathcal{S}$ during the setup.
- For $i \in \mathcal{I}$ and $j \in \mathcal{H}$, add $(Z, K'_{\alpha,j})$ to a corrupt party P_i's view if $i = j - 1$.

Phase 3 Simulation. For $i \in \mathcal{I}$ and $j \in \mathcal{H}$, add y to a corrupt party P_i's view if $i = j + 1$.

Below, we argue that the views of corrupt parties in the real and ideal world are indistinguishable via a series of intermediate hybrids.

-Hyb_0: Same as the real-world execution.

-Hyb_1: Same as the previous hybrid except that c^*_j for $j \in \mathcal{H}$ and $j \neq \mathsf{q}$ is computed as $\mathcal{S}_{\mathtt{AHE}}(\mathtt{pk}, 0)$.

This is in contrast to the previous hybrid where c^*_j for $j \in \mathcal{H}$ and $j \neq \mathsf{q}$ is computed as $\mathtt{ScalMul}(\mathtt{pk}, \mathtt{Enc}(\mathtt{pk}, 0), x_j)$. Indistinguishability follows from circuit privacy of the $\mathtt{AHE}$.

-Hyb_2: Same as the previous hybrid except that ct_j for $j \in \mathcal{H}$ is computed as $\mathcal{S}_{\mathtt{AHE}}(\mathtt{pk}, m)$, where $m = y$ if $x_1 = \mathsf{q} \leq j$ and $m = 0$ otherwise.

This is in contrast to the previous hybrid where ct_j for $j \in \mathcal{H}$ is computed as $\mathtt{Add}(\mathtt{pk}, \mathtt{Enc}(\mathtt{pk}, 0), \mathtt{Enc}(\mathtt{pk}, m))$, where $m = y$ if $x_1 = \mathtt{q} \leq j$ and $m = 0$ otherwise. Indistinguishability follows from circuit privacy of the $\mathtt{AHE}$.

-Hyb$_3$: Same as Hyb$_2$, except for the way in which $K'_{\alpha,j}$ for $j \in \mathcal{H}, \alpha \in [\lambda]$ is computed – For $j \neq j_{\min}$, $K'_{\alpha,j}$ is chosen uniformly at random, whereas for $j = j_{\min}$, it is set such that $K'_{\alpha,j_{\min}} + \sum_{i=1}^{j_{\min}-1} K_{\alpha,i}^{(z_\alpha)} = K_\alpha^{(z_\alpha)}$.

This is in contrast to the previous hybrid where $K'_{\alpha,j}$ is computed as $K'_{\alpha,j} \leftarrow K'_{\alpha,j+1} + K_{\alpha,j}^{(z_\alpha)}$. Since $K_{\alpha,j}^{(z_\alpha)}$ is distributed uniformly at random (conditioned on $\Sigma_{k=1}^{n} K_{\alpha,k}^{(z_\alpha)} = K_\alpha^{(z_\alpha)}$), this is indistinguishable from the previous hybrid.

-Hyb$_4$: Same as Hyb$_3$, except that $(\mathtt{GC}_{\mathtt{Dec}}, \vec{K})$ is computed as $\mathtt{GC}_{\mathtt{Dec}} \leftarrow \mathtt{simGC}(1^\lambda, \theta(C), y, \vec{K})$, where $\vec{K} = \{K_1, \ldots, K_\lambda\}$ is sampled at random and $\theta(C)$ denotes the topology of the circuit computing the decryption of an $\mathtt{AHE}$ input ciphertext.

This is in contrast to Hyb$_3$ where $\mathtt{GC}_{\mathtt{Dec}}$ is computed as $\mathtt{GC}_{\mathtt{Dec}} \leftarrow \mathtt{garble}(C_{\mathtt{sk}}, \{K_\alpha^{(0)}, K_\alpha^{(1)}\}_{\alpha\in[\lambda]}, 1^\lambda)$ where $\{K_\alpha^{(0)}, K_\alpha^{(1)}\}_{\alpha\in[\lambda]}$ is sampled at random and $\vec{K} = \{K_\alpha^{(z_\alpha)}\}_{\alpha\in[\lambda]}$ It follows from privacy of the garbling scheme (see Definition 5) that Hyb$_4$ is indistinguishable from Hyb$_3$.

Since Hyb$_4$ corresponds to the ideal execution and every pair of consecutive hybrids are indistinguishable, this completes the proof that the views of corrupt parties in the real and ideal world are indistinguishable for the case when $P_1 \in \mathcal{I}$.

Next, we describe the simulation for the case when $P_1 \in \mathcal{H}$. The main difference is that in this case we rely on privacy of garbling with respect to $\mathtt{GC}_i$ ($i \in \mathcal{I}$). On the other hand, simulation of $\mathtt{GC}_{\mathtt{Dec}}$ is not relevant here as the adversary (who does not corrupt P_1) does not have access to $\mathtt{GC}_{\mathtt{Dec}}$ in the real-world execution of the protocol.

Setup Simulation.

- For each $i \in \mathcal{I}$: Sample $\mathtt{pk}, r_{i,\mathtt{Enc}}$ and compute $c'_i = \mathtt{Enc}(\mathtt{pk}, 0; r_{i,\mathtt{Enc}})$. Run $\mathtt{GC}_i \leftarrow \mathtt{simGC}(1^\lambda, \theta(C), c'_i, \{\widetilde{K}_1, \ldots, \widetilde{K}_\lambda\})$, where $\mathtt{simGC}$ denotes the simulator of the garbling scheme, $\{\widetilde{K}_1, \ldots, \widetilde{K}_\lambda\}$ are chosen at random and $\theta(C)$ denotes the topology of the circuit $C_{i,\mathtt{pk},r_{i,\mathtt{Enc}}}$ described in $\Pi_{\mathtt{sel}}$ (Figure 4.1). Note that the topology of the circuit is independent of the hard-coded values.
- For each $i \in \mathcal{I}$, sample $\left(\{K_{\alpha,i}^{(0)}, K_{\alpha,i}^{(1)}\}_{\alpha\in[\lambda]}\right)$ at random.

The view of corrupted P_i ($i \geq 2$) constructed by $\mathcal{S}$ at this stage comprises of $\left(\mathtt{pk}, \mathtt{GC}_i, \{K_{\alpha,i}^{(0)}, K_{\alpha,i}^{(1)}\}_{\alpha\in[\lambda]}\right)$.

Phase 1 Simulation. On behalf of each honest P_j ($j \in \mathcal{H}$), compute $ct_j \leftarrow \mathcal{S}_{\mathtt{AHE}}(\mathtt{pk}, 0)$, where $\mathcal{S}_{\mathtt{AHE}}$ refers to the simulator of the $\mathtt{AHE}$ for circuit privacy (Definition 6). . For $i \in \mathcal{I}$ and $j \in \mathcal{H}$, add $(\mathbf{K} = \{\widetilde{K}_1, \ldots, \widetilde{K}_\lambda\}, ct_j)$ to a corrupt party P_i's view if $i = j + 1$.

Phase 2 Simulation. Recall that for each $i \in \mathcal{I}$, $\mathcal{S}$ knows x_i and c'_i (during the simulation of the setup) and can therefore compute c^*_i.

- Compute Z by homomorphic addition of $ct_{j_{\max}}$ and each c^*_k for $k \in [j{+}1, n]$. Parse $Z = (z_1, z_2, \ldots, z_\lambda)$.
- For each $\alpha \in \lambda$ and $j \in \mathcal{H}$: Sample $K'_{\alpha,j}$ at random.
- For $i \in \mathcal{I}$ and $j \in \mathcal{H}$, add $(Z, K'_{\alpha,j})$ to a corrupt party P_i's view if $i = j - 1$.

Phase 3 Simulation. For $i \in \mathcal{I}$ and $j \in \mathcal{H}$, add y to a corrupt party P_i's view if $i = j + 1$.

Below, we argue that the views of corrupt parties in the real and ideal world are indistinguishable via a series of intermediate hybrids.

Hyb_0: Same as the real-world execution.

Hyb_1: Same as the previous hybrid, except that y is sent by P_1 in Phase 3 directly, without evaluating $\mathtt{GC}_{\mathtt{Dec}}$.

This is indistinguishable from the previous hybrid as the adversary's view is identical.

Hyb_2: Same as the previous hybrid except that $\mathtt{GC}_i$ for $i \in \mathcal{I}$ outputs $\mathtt{Enc}(\mathtt{pk}, 0; r_{i,\mathtt{Enc}})$.

This hybrid differs from the previous one only if $i = \mathsf{q}$ holds. In such a case, c'_i corresponds to an encryption of 1 in the previous hybrid (as opposed to encryption of 0 in Hyb_2). This is indistinguishable from the previous hybrid due to CPA security of $\mathtt{AHE}$.

Hyb_3: Same as the previous hybrid except that honest parties P_j ($j \in \mathcal{H}$) compute c'_j as $c'_j \leftarrow \mathtt{Enc}(\mathtt{pk}, 0; r_{j,\mathtt{Enc}})$.

This hybrid differs from the previous one only if $j = \mathsf{q}$ holds. In such a case, c'_j corresponds to an encryption of 1 in the previous hybrid (as opposed to encryption of 0 in Hyb_3). This is indistinguishable from the previous hybrid due to CPA security of $\mathtt{AHE}$.

Hyb_4: Same as the previous hybrid except that c^*_j for $j \in \mathcal{H}$ is computed as $\mathcal{S}_{\mathtt{AHE}}(\mathtt{pk}, 0)$.

This is in contrast to the previous hybrid where c^*_j is computed as $c^*_j \leftarrow \mathtt{ScalMul}(\mathtt{pk}, \mathtt{Enc}(\mathtt{pk}, 0), x_j)$. Indistinguishability follows from circuit privacy

of the $\mathtt{AHE}$.

-Hyb_5: Same as the previous hybrid except that ct_j for $j \in \mathcal{H}$ is computed as $\mathcal{S}_{\mathtt{AHE}}(\mathtt{pk}, 0)$.

This is in contrast to the previous hybrid where ct_j is computed as $\mathtt{Add}(\mathtt{pk}, \mathtt{Enc}(\mathtt{pk}, 0), \mathtt{Enc}(\mathtt{pk}, 0))$. Indistinguishability follows from circuit privacy of the $\mathtt{AHE}$.

-Hyb_6: Same as Hyb_5, except that $\mathtt{GC}_i$ for $i \in \mathcal{I}$ is computed as $\mathtt{GC}_i \leftarrow \mathtt{simGC}(1^\lambda, \theta(C), c'_i, \{\widetilde{K}_1, \ldots, \widetilde{K}_\lambda\})$, where $\{\widetilde{K}_1, \ldots, \widetilde{K}_\lambda\}$ are chosen at random and $\theta(C)$ denotes the topology of the circuit $C_{i,\mathtt{pk},r_{i,\mathtt{Enc}}}$ described in $\Pi_{\mathtt{sel}}$ (Figure 4.1).

This is in contrast to Hyb_5 where $\mathtt{GC}_i, \{\widetilde{K}_1, \ldots, \widetilde{K}_\lambda\}$ are computed as $\mathtt{GC}_i \leftarrow \mathtt{garble}(C_{i,\mathtt{pk},r_{i,\mathtt{Enc}}}, \{\widetilde{K}_\alpha^{(0)}, \widetilde{K}_\alpha^{(1)}\}_{\alpha\in[\lambda]}, 1^\lambda)$. Here, $\{\widetilde{K}_\alpha^{(0)}, \widetilde{K}_\alpha^{(1)}\}_{\alpha\in[\lambda]}$ is sampled at random and $\{\widetilde{K}_1, \ldots, \widetilde{K}_\lambda\} = \{K_\alpha^{(b_\alpha)}\}_{\alpha\in[\lambda]}$ where $\mathtt{q} = (b_1, \ldots, b_\lambda)$. It follows from privacy of the garbling scheme (see Definition 5) that Hyb_6 is indistinguishable from Hyb_5.

Since Hyb_6 corresponds to the ideal execution and every pair of consecutive hybrids are indistinguishable, this completes the proof that the views of corrupt parties in the real and ideal world are indistinguishable for the case when $P_1 \in \mathcal{H}$.

References

ABJ+19. Prabhanjan Ananth, Saikrishna Badrinarayanan, Aayush Jain, Nathan Manohar, and Amit Sahai. From FE combiners to secure MPC and back. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 199–228. Springer, Heidelberg, December 2019.

AJL+12. Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 483–501. Springer, Heidelberg, April 2012.

BGI+14. Amos Beimel, Ariel Gabizon, Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, and Anat Paskin-Cherniavsky. Non-interactive secure multiparty computation. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 387–404. Springer, Heidelberg, August 2014.

BGT13. Elette Boyle, Shafi Goldwasser, and Stefano Tessaro. Communication locality in secure multi-party computation - how to run sublinear algorithms in a distributed setting. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 356–376. Springer, Heidelberg, March 2013.

BGW88. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

BHR12. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

BJPY18. Elette Boyle, Abhishek Jain, Manoj Prabhakaran, and Ching-Hua Yu. The bottleneck complexity of secure multiparty computation. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *ICALP 2018*, volume 107 of *LIPIcs*, pages 24:1–24:16. Schloss Dagstuhl, July 2018.

BV11. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In Rafail Ostrovsky, editor, *52nd FOCS*, pages 97–106. IEEE Computer Society Press, October 2011.

Can00. Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000.

CCD88. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th ACM STOC*, pages 11–19. ACM Press, May 1988.

CLTV15. Ran Canetti, Huijia Lin, Stefano Tessaro, and Vinod Vaikuntanathan. Obfuscation of probabilistic circuits and applications. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part II*, volume 9015 of *LNCS*, pages 468–497. Springer, Heidelberg, March 2015.

Cou19. Geoffroy Couteau. A note on the communication complexity of multiparty computation in the correlated randomness model. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 473–503. Springer, Heidelberg, May 2019.

DFH12. Ivan Damgård, Sebastian Faust, and Carmit Hazay. Secure two-party computation with low communication. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 54–74. Springer, Heidelberg, March 2012.

DI06. Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 501–520. Springer, Heidelberg, August 2006.

DIK+08. Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam Smith. Scalable multiparty computation with nearly optimal work and resilience. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 241–261. Springer, Heidelberg, August 2008.

DNPR16. Ivan Damgård, Jesper Buus Nielsen, Antigoni Polychroniadou, and Michael Raskin. On the communication required for unconditionally secure multiplication. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 459–488. Springer, Heidelberg, August 2016.

EOYN21. Reo Eriguchi, Kazuma Ohara, Shota Yamada, and Koji Nuida. Non-interactive secure multiparty computation for symmetric functions, revisited: More efficient constructions and extensions. In Tal Malkin and Chris Peikert, editors, *CRYPTO*, 2021.

FKLS20. Rex Fernando, Ilan Komargodski, Yanyi Liu, and Elaine Shi. Secure massively parallel computation for dishonest majority. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC*, 2020.

GMRW13. S. Dov Gordon, Tal Malkin, Mike Rosulek, and Hoeteck Wee. Multi-party computation of polynomials and branching programs without simultaneous interaction. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 575–591. Springer, Heidelberg, May 2013.

GMW87. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

HIJ+16. Shai Halevi, Yuval Ishai, Abhishek Jain, Eyal Kushilevitz, and Tal Rabin. Secure multiparty computation with general interaction patterns. In Madhu Sudan, editor, *ITCS 2016*, pages 157–168. ACM, January 2016.

HIKR18. Shai Halevi, Yuval Ishai, Eyal Kushilevitz, and Tal Rabin. Best possible information-theoretic MPC. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018, Part II*, volume 11240 of *LNCS*, pages 255–281. Springer, Heidelberg, November 2018.

HLP11. Shai Halevi, Yehuda Lindell, and Benny Pinkas. Secure computation on the web: Computing without simultaneous interaction. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 132–150. Springer, Heidelberg, August 2011.

IKM+13. Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 600–620. Springer, Heidelberg, March 2013.

IMO18. Yuval Ishai, Manika Mittal, and Rafail Ostrovsky. On the message complexity of secure multiparty computation. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018, Part I*, volume 10769 of *LNCS*, pages 698–711. Springer, Heidelberg, March 2018.

IP07. Yuval Ishai and Anat Paskin. Evaluating branching programs on encrypted data. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 575–594. Springer, Heidelberg, February 2007.

LNO13. Yehuda Lindell, Kobbi Nissim, and Claudio Orlandi. Hiding the input-size in secure two-party computation. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 421–440. Springer, Heidelberg, December 2013.

NN01. Moni Naor and Kobbi Nissim. Communication preserving protocols for secure function evaluation. In *33rd ACM STOC*, pages 590–599. ACM Press, July 2001.

QWW18. Willy Quach, Hoeteck Wee, and Daniel Wichs. Laconic function evaluation and applications. In Mikkel Thorup, editor, *59th FOCS*, pages 859–870. IEEE Computer Society Press, October 2018.

Reg05. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

Yao82. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.

Yao86. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.