MPC with Low Bottleneck-Complexity: Information-Theoretic Security and More

Hannah Keller[1], Claudio Orlandi[1], Anat Paskin-Cherniavsky[2], and Divya Ravi[1] *

[1] Aarhus University, Denmark,{hkeller,orlandi,divya}@cs.au.dk
[2] Ariel University, Israel, anatpc@ariel.ac.il

Abstract. The bottleneck-complexity ($\mathcal{BC}$) of secure multiparty computation (MPC) protocols is a measure of the maximum number of bits which are sent and received by any party in protocol. As the name suggests, the goal of studying $\mathcal{BC}$-efficient protocols is to increase overall efficiency by making sure that the workload in the protocol is somehow "amortized" by the protocol participants.
Orlandi et al. [ORS22] initiated the study of $\mathcal{BC}$-efficient protocols from simple assumptions in the correlated randomness model and for semi-honest adversaries. In this work, we extend the study of [ORS22] in two primary directions: **(a)** to a larger and more general class of functions and **(b)** to the information-theoretic setting.
In particular, we offer semi-honest secure protocols for the useful function classes of abelian programs, 'read-k' non-abelian programs, and 'read-k' generalized formulas.
Our constructions use a novel abstraction, called *incremental function secret-sharing* (IFSS), that can be instantiated with unconditional security or from one-way functions (with different efficiency trade-offs).

1 Introduction

Secure Multi-party Computation (MPC) [Yao86,GMW87,BGW88,CCD88], allows a set of mutually distrusting parties to perform a joint computation of their private inputs in a secure way, which essentially means that no adversary corrupting a subset of parties can learn more information than the output of the joint computation (*privacy*), nor can they affect the *correctness* of the output (other than by choosing their own inputs).

The complexity of MPC protocols is most commonly analyzed in terms of three fundamental metrics, namely communication complexity (that measures the total number of bits communicated in the protocol), round complexity (number of sequential interactions in the protocol) and computation complexity (that captures the computational resources parties need to execute the protocol steps). In this paper, we focus on a more fine-grained, comparitively less-explored metric called *bottleneck complexity ($\mathcal{BC}$)* which was introduced by Boyle et al. [BJPY18]. This metric, which can be informally defined as the *maximum communication complexity of any party* captures the load-balancing aspect of MPC protocols – for e.g. a protocol where everyone sends a bit to a central party would have $O(n)$ $\mathcal{BC}$ (incurred by the central party, where n denotes the number of parties) as opposed to a protocol where everyone sends a single bit to its neighbour in a chain-like fashion, which has $O(1)$ $\mathcal{BC}$. Notably, both these protocols have the same communication complexity but the communication in the latter is more balanced among the parties as captured by its lower bottleneck complexity.

The works of [BJPY18,ORS22] focused on designing MPC protocols with bottleneck complexity sublinear in the number of parties, which is particularly interesting for large-scale settings where n is huge. [BJPY18] presented a FHE-based compiler that transforms insecure protocols into secure protocols while preserving the bottleneck complexity. However, FHE is still relatively inefficient, and is only known under a more limited set of assumptions - roughly, variants of LWE. In light of this, [ORS22] initiated the study of designing protocols

* Funded 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).

with low bottleneck complexity in the preprocessing model, under minimal computational assumptions (such as one-way functions and linearly homomorphic encryption, which can in turn be based on traditional assumptions such as discrete logarithm and factoring).

In this work, we extend the study of [ORS22] in two primary directions: **(a)** to a larger and more general class of functions and **(b)** to the information-theoretic setting. We additionally consider a more extended notion of g-$\mathcal{BC}$-efficiency to capture protocols which have $\mathcal{BC}$ of $\mathsf{poly}(g(n), \lambda)$, where λ denotes the security parameter[3]. More specifically, [ORS22] focused on protocols with $O(1)$-$\mathcal{BC}$-efficiency (i.e. with $\mathsf{poly}(\lambda)$ $\mathcal{BC}$, independent of n) and log-$\mathcal{BC}$-efficiency (i.e. with $\mathsf{poly}(\log(n), \lambda)$ $\mathcal{BC}$); while we consider a more general notion of g-$\mathcal{BC}$ efficiency, where g is any *sublinear* function. This allows us to work with a somewhat extended parameter setting – Consider a function $f(x_1, ..., x_n)$ where each x_i has ℓ bits, and the (common) output is z bits. The notions of $\mathcal{BC}$-efficiency become meaningful only if these parameters ℓ and z are typically small. In the prior work of [ORS22], these are assumed to be constant or polylogarithmic in n. In this work we extend our quest to settings where ℓ and z are sublinear in size ($o(n)$), as this would still allow for constructions satisfying the extended notion of $\mathcal{BC}$-efficiency. Moreover, the constructions of [BJPY18,ORS22] have $\mathcal{BC}$ that scales with the security parameter λ (where λ is typically $\omega(\log(n))$) in computational settings), which is avoided by our information-theoretic constructions.

Related Work. The most relevant work to ours is [BJPY18,ORS22] (whose results we discuss above). There are several works in the MPC literature that focus on optimizing communication complexity, some of which we mention below. The works of [Cou19,DNPR16,IKM$^+$13] focus on designing communication-efficient protocols in the information-theoretic setting with correlated randomness. Interestingly, the notion of bottleneck complexity and communication complexity are the same for the two-party setting. The work of [NN01] presented a compiler that transforms an insecure protocol to secure one while preserving communication complexity. [DI06,DIK$^+$08,QWW18,ABJ$^+$19] focus on optimizing communication complexity related to the circuit size.

The constructions of [HIJ$^+$16,HLP11,GMRW13] involve a chain-like interaction pattern (similar to our constructions). However, these constructions achieve a weaker notion of security (namely, residual security) as they are restricted to a single chain traversal (unlike our constructions which typically involve multiple traversals over chain). The efficient non-interactive multiparty computation (NIMPC) constructions in [HIKR18,EOYN21,BGI$^+$14] also achieve this weaker security.

Our goal of minimizing bottleneck complexity is somewhat similar in spirit to the massive parallel computation model of [FKLS20,FGKS22] which focuses on minimizing the storage and communication of servers. The works of [BGT13] and [IMO18] design protocols that optimize metrics that are closely related to bottleneck complexity (namely, communication locality and message complexity).

For further related work, we refer to references therein.

1.1 Our Contribution

Our main contribution is constructing $\mathcal{BC}$-efficient protocols in the correlated randomness model for various interesting function classes. Further, we introduce a new primitive, namely *Incremental Function Secret Sharing* (IFSS), which not only serves as a neat abstraction of $\mathcal{BC}$-efficient computation, but also allows us to cast our constructions in a generalized framework that captures both computational and information-theoretic variants.

All our constructions are secure against a semi-honest (passive) adversary who can corrupt up to $n-1$ among the n parties. Our computational constructions are based on garbling schemes, which rely on one-way functions. Our information-theoretic constructions (satisfying perfect security) are a $\mathcal{BC}$-friendly extension of the OTTT (one-time truth-tables) construction from [IKM$^+$13].

We elaborate on our contributions below.

[3] For information-theoretic protocols with perfect security where there is no dependency on λ, g-$\mathcal{BC}$ efficiency refers to $\mathcal{BC}$ of $\mathsf{poly}(g(n))$.

New primitive: Incremental Function Secret Sharing (IFSS). In Section 4, we introduce a new primitive, namely, Incremental Function Secret Sharing (IFSS), which essentially allows a set of parties to evaluate a hidden function on a joint public input. This tool is a clean abstraction of the core ideas of $\mathcal{BC}$-efficient evaluation in our constructions.

At a high-level, this primitive can be viewed as a variant of function secret sharing (which additively shares a function among a set of evaluators, enabling them to compute output shares which can be aggregated to obtain the output), with the difference that the output shares are aggregated incrementally on a chain, in a $\mathcal{BC}$-efficient manner. IFSS can be instantiated with garbled circuits or one-time truth tables (OTTT), enabling us to unify our computational and information-theoretic variants. We believe this primitive to be of independent interest and a useful building block for $\mathcal{BC}$-efficient protocols.

Abelian Programs. Recall that an abelian program h can be expressed as $h(X_1, \ldots, X_n) = f(\sum_{i=1}^{n} X_i)$ for some $f : G \to \{0,1\}$, where G denotes an abelian group. In Section 5, we use our IFSS primitive to generalize the approach of [ORS22] that constructs $\mathcal{BC}$-efficient (computational) protocols for abelian programs. Plugging in the information-theoretic OTTT-based instantiation of IFSS yields an information-theoretic $\mathcal{BC}$-efficient protocol for abelian programs with $\mathcal{BC}$ of $O(\log |G|)$. For completeness, we additionally demonstrate how using the garbled-circuit based instantiation of IFSS results in the computational protocol of [ORS22].

As an interesting application of $\mathcal{BC}$-efficient abelian programs, we demonstrate how it could be used to compute the maximum among n values as $y = \max(X_1, X_2, \ldots, X_n)$ in a $\mathcal{BC}$-efficient manner. This can be extended to compute $f(\max(X_1, X_2, \ldots, X_n))$, where f is any arbitrary function.

'Read-k' Non-Abelian Programs. Briefly, a non-abelian program extends the notion of an abelian program to non-abelian groups. Here, $h(x_1, \ldots, x_n) : \{D\}^n \to \{0,1\}$ is represented as $h(x) = g(\pi_{1,x_{i_1}} \cdot \ldots \cdot \pi_{t,x_{i_t}})$, where each $\pi_{j,x_{i_j}}$ is an element of a group G that depends only on j, x_{i_j} (where $i_j \in \{1, \ldots, n\}$), and $f : G \to \{0,1\}$[4]. In a read-k program, group elements depending on some x_i appear up to k times in the above representation.

In Section 6, we present a $\mathcal{BC}$-efficient protocol for any function f, that can be represented as a read-k non-abelian program over a group G. The computational and information-theoretic variants of these constructions incur a $\mathcal{BC}$ of $O(\log |G|(\lambda + k))$ and $O(k \log |G|)$ respectively and will therefore have sublinear $\mathcal{BC}$ as long as each of the parameters k and $\log |G|$ are 'sufficiently small'. More specifically, we can allow $k \log |G|$ to be of size $o(n)$. Even for e.g., $|G| = 2^{n^\epsilon}, k = O(1)$ for $\epsilon \in (0,1)$, we obtain sublinear $\mathcal{BC}$.

'Tree-based' read-k generalized formulas. In Section 7,we present a $\mathcal{BC}$-efficient protocol for any function f that can be represented as a 'tree-like' formula, which may have multi-input (and output) gates. More concretely, nodes in this formula are either inputs X_i (which may belong to some finite group, not necessarily boolean domain), and gates with 2 inputs that output a single output [5]. The inputs, intermediate outputs (which are inputs to other gates) and the output are assumed to be bounded by ℓ bits (where $\ell = \log |G|$ in our constructions). For a formula that is read-k (i.e. each input variable appears at most k times) and has depth d [6], our computational and information-theoretic variants result in $\mathcal{BC}$ of $O(k \cdot d \cdot \ell \cdot \lambda)$ and $O(k \cdot d \cdot \ell)$ respectively.

As long as the above parameters of k, d, and ℓ are 'sufficiently small' (i.e. $k \cdot d \cdot \ell$ is of size $o(n)$), the protocols remain $\mathcal{BC}$-efficient. Even with these restrictions, such formulas are quite expressive. Notably, we allow for 'generalized' gates in terms of the functions they compute – the above restrictions are only with respect to the size of the inputs and outputs of these gates but the structure of the functions computed by these gates may be quite complex. For example, consider a generalized formula of depth $O(\log(n))$, and

[4] Note that unlike abelian programs, some π_{x_i} depending on x_i may crucially appear more than once, as the group G is not commutative.

[5] The construction is actually more general, and could allow for 'generalized gates' with larger fan-in, but we stick with 2 for simplicity.

[6] Note that for balanced trees, $d = \log_2(k \cdot n)$, which is sublinear in n.

$k = \ell = n^{0.4}$, where each gate (having two inputs and an output of length $\ell = n^{0.4}$), evaluates a function with circuit complexity of $\Omega(2^{n^{0.5}})$. The $\mathcal{BC}$ complexity of the protocol above only would be $\tilde{O}(n^{0.8})$ [7] for our information theoretic implementation.

Lastly, we point out that while our information-theoretic constructions have better $\mathcal{BC}$ than the computational variants, the size of the correlated randomness for our information-theoretic constructions grows exponentially with the number of parties (due to the OTTT approach). However, the $\mathcal{BC}$ still remains sublinear in the number of parties for all our constructions.

In Section 2, we compare the expressiveness of the above function classes of abelian, non-abelian programs and tree-based formulas.

Open Problems. It remains an open question to determine the complete characterization of functions for which $\mathcal{BC}$-efficient protocols exist. In fact, since $\mathcal{BC}$-efficient protocols are known to be impossible for general functions even when no security is required [BJPY18], it would also be interesting to understand which functions can be computed in the clear with low bottleneck complexity.

1.2 Technical Overview

Our constructions have the following common two-step approach: **(1)** First, the private inputs are aggregated in a $\mathcal{BC}$-efficient way to obtain a joint common public input that 'hides' the private inputs. **(2)** Next, we consider an augmented version of the function f, say f' (that may be required to be kept secret) such that evaluating f' on the common public input essentially corresponds to an evaluation of f. The evaluation of f' is carried out via IFSS.

Overview of IFSS. Before describing details of our protocols, we give a high-level overview of the IFSS primitive. In a nutshell, IFSS allows a set of parties to evaluate a hidden function f on a common public input x such that nothing beyond $f(x)$ is revealed (as long as one of the evaluators is honest). This evaluation is done in an incremental manner, where each party computes its 'share' and these shares are aggregated over a chain. In the garbled-circuit based instantiation, these 'shares' are additive shares of the label corresponding to the common input x. Once this label is reconstructed (via aggregation over chain), the garbled circuit computing f (given as part of the setup) is evaluated to compute the output. This is the crux of $\mathcal{BC}$-efficient evaluation in the constructions of [ORS22], which satisfy computational security.

For the information-theoretic instantiation, we use an approach based on secret-sharing the truth-table inspired by the one-time truth table (OTTT) protocol of [IKM+13]. As already noted by [BGI16], this leads to information-theoretic FSS. We detail the construction and show how it fits the IFSS framework.

In this protocol, parties are given an additive sharing of the (permuted) truth table of the function being evaluated, as a part of the correlated randomness setup. Roughly speaking, the parties first identify the relevant entry of the truth table (using their input and correlated randomness) i.e. the one that corresponds to the correct output. The pointer to this entry can be interpreted as the common input of the IFSS. Now, the evaluation is nothing but aggregating the additive shares of the relevant entry (determined by this common input), which can be done in a chain-like fashion to maintain $\mathcal{BC}$-efficiency. This is the main idea of the information-theoretic instantiation of IFSS.

Next, we describe our constructions. Note that for protocols to be $\mathcal{BC}$-efficient, the interaction involved in the above outlined common two-step approach must satisfy the following two properties: **(a)** Each intermediate value that is communicated must be 'small'. **(b)** Privacy of the inputs must be maintained.

Abelian Programs. The structure of abelian programs (say $h(x_1, \ldots, x_n) = f(\sum_{i=1}^{n} x_i)$) is such that it naturally supports property **(a)**.

[7] $\tilde{O}$ ignores logarithmic factors.

This is because the sum of inputs can be computed incrementally in a chain-like fashion with the property that the size of the intermediate sums does not blow up. However, to satisfy **(b)**, the protocol of [ORS22] makes parties aggregate their masked inputs instead (using masks received as part of setup) to compute a masked sum (say $z = y + R$, where y denotes the sum of inputs and R denotes the mask). Generalizing their construction, we view this 'masked sum' as the common input, and use IFSS for evaluation. More specifically, we consider a (private) augmented function $f'_R(z) = f(z - R)$ (with secret R hard-coded), which first unmasks this masked sum to retrieve the sum y, upon which h is computed. We use IFSS to compute f', yielding computational and information-theoretic $\mathcal{BC}$-efficient protocols (depending on whether the IFSS is instantiated using the garbling-based or OTTT based approach).

'Read-k' Non-Abelian Programs. Similar to abelian programs, non-abelian programs (say, $h(x) = f(\pi_{1,x_{i_1}} \cdot \ldots \cdot \pi_{t,x_{i_t}})$) support property **(a)** as the input value to f (i.e. $\pi_{1,x_{i_1}} \cdot \ldots \cdot \pi_{t,x_{i_t}}$) can be computed incrementally in a chain-like fashion (where party i_1 forwards $\pi_{1,x_{i_1}}$ to i_2 who computes $\pi_{1,x_{i_1}} \cdot \pi_{2,x_{i_2}}$ and forwards this value to i_3 and so on) while making sure that the intermediate values remain 'small'. To maintain property **(b)**, the aggregation could be done over masked inputs instead. However, unlike the case of abelian groups (which is commutative), we need to be slightly more careful in case of non-abelian groups (which may be non-commutative) to ensure that this aggregation of masked inputs happens in a specific order. In our protocol, the aggregated common input corresponds to $\left(\mathsf{r}_t \cdot \ldots (\mathsf{r}_2 \cdot (\mathsf{r}_1 \cdot \pi_{1,x_{i_1}}) \cdot \pi_{2,x_{i_2}}) \cdot \ldots \cdot \pi_{t,x_{i_t}}\right)$; accordingly each party i_j must compute its intermediate value by using its random value as a prefix and $\pi_{j,x_{i_j}}$ as a suffix to the intermediate value received from its neighbour on the chain. Once, parties have computed this common public input, we use IFSS to compute the augmented function f', where f' first uses a (secret hard-coded) prefix $\mathsf{r}_1^{-1} \cdot \mathsf{r}_2^{-1} \ldots \cdot \mathsf{r}_t^{-1}$ to unmask this common input and then compute f.

'Tree-based' read-k generalized formulas. Next, consider the case of 'tree-like' formulas. Consider one of the 'generalized gates' say $f(x_1, \ldots, x_{\mathsf{m}})$ (whose number of inputs and output size is 'sufficiently small' [8] but could have any arbitrarily complicated structure). Unlike the previous cases, we cannot exploit the structure of the function to support incremental aggregation (that supports property **(a)**). Instead, every party involved in f must compute its masked input (using a random value given as part of the setup) and communicate it in a chain-like fashion *without* any incremental computation. These masked inputs are simply appended (therefore the size of the intermediate values grows in this case) and this set of masked values forms the aggregated common input. Note that the size of this aggregated common input grows with the number of inputs to this 'generalized' gate, which brings in the need for restricting the size and number of inputs to these gates to be 'small' (i.e. $o(n)$) for sublinear $\mathcal{BC}$.

Once the common input is determined, we proceed to evaluation. For this, we consider an instance of IFSS for each 'generalized' gate and combine the intermediate outputs in a tree-like fashion. For simplicity, consider a gate at the first level, computing $f_{(1,2)}$ that takes two leaf nodes (corresponding to inputs x_1 and x_2) as input. As mentioned previously, the aggregate common input corresponds to $z = z_1 || z_2$, where $z_i = x_i + \mathsf{r}_i$ for $i \in [2]$, where r_i are random masks given during setup. An instance of IFSS with hidden function $f'_{(1,2)}$ (that has r_i values as hard-coded inputs) and single common input z is initiated, that first unmasks the random values from the set of masked inputs z and then computes the output of $f_{(1,2)}$. This instance involves only the subset of parties holding one of these inputs x_1 or x_2 as evaluators, not the set of all parties. Note that this does not violate security because even if one of the parties contributing an input to $f_{(1,2)}$ is honest, by IFSS security, the parties will not learn anything except of the output of $f_{(1,2)}$. Otherwise, if all parties are corrupted, they know all their inputs anyway, so there is nothing to hide. More generally, the IFSS instance corresponding to a root of a subtree involves only the parties whose input is one of the leaf nodes of this subtree. This is crucial to maintain $\mathcal{BC}$-efficiency.

However, this approach would result in parties learning the output of $f_{(1,2)}$ which may not necessarily be leaked by the output of f. Therefore, instead of computing the original $f_{(1,2)}$, we compute the modified function $f'_{(1,2)}$ that 'masks' the output of $f_{(1,2)}$ with a random mask $\mathsf{r}_{(1,2)}$ chosen during setup. We ensure correctness of computation by defining the functions at the levels above accordingly – for instance, consider

[8] More specifically, these parameters are bounded by ℓ such that $k \cdot d \cdot \ell$ is of size $o(n)$.

another function $f_{(3,4)}$ at level 1 (with its similarly defined $f'_{(3,4)}$ that 'masks' the output of $f_{(3,4)}$ with random mask $\mathsf{r}_{(3,4)}$). Suppose the 'tree-like' formula had a function $f_{(1,4)}$ at level 2 that is supposed to take as input $y_{(1,2)}$ and $y_{(3,4)}$ (respectively the outputs of $f_{(1,2)}$ and $f_{(3,4)}$). We now define $f'_{(1,4)}$ as the function with hard-coded masks $\mathsf{r}_{(1,2)}$,$\mathsf{r}_{(3,4)}$ and $\mathsf{r}_{(1,4)}$ that receives instead masked inputs $(y_{(1,2)} + \mathsf{r}_{(1,2)})$ and $(y_{(3,4)} + \mathsf{r}_{(3,4)})$, unmasks them, evaluates $f_{(1,4)}$, and finally masks the output with $\mathsf{r}_{(1,4)}$ (unless this node corresponds to the root). The evaluations are done level-by-level in an upward fashion until the function corresponding to the root of the tree is computed. This demonstrates how the outputs of various instances of IFSS computations are combined; completing the high-level description of this construction.

Lastly, we point that in our constructions, the common input used in the IFSS could be dictated by the adversary and still 'unknown' to the honest party when the evaluation of IFSS begins. For e.g. consider the case of abelian programs, where the common input is the sum of masked inputs and computed over a forward-pass of the chain. Suppose the adversary corrupts a set of parties at the end of the chain and the evaluation of IFSS is executed over the subsequent backward pass of the chain. In such a case, this common input is in some sense still 'uncommitted' (as the adversary can consider various versions of the common input and try to recompute the IFSS incremental evaluations in her head). Even though the adversary is passive, she can try to learn more information by trying to obtain multiple evaluations of IFSS corresponding to different common inputs (referred to as a residual function attack); which would breach security [9]. Security of IFSS does not help in this case as it holds only if all evaluators agree on the common input. However, our constructions ensure that the common input gets 'committed' as soon as the backward pass reaches the first honest evaluator. The incremental computation by this honest evaluator would 'fix' the common input (in a way that it is not possible to recompute evaluations on other common inputs any further), which creates the effect of 'fixing' the corrupt evaluators' inputs. We refer to respective technical sections for further details.

2 Comparison of the function classes

In this section, we discuss what kind of functions are captured by the function classes considered in this work.

Abelian versus Non-Abelian Programs. We observe that indeed non-abelian programs appear to be more expressive than abelian programs within our $\mathcal{BC}$ constraints. As a nice simple example, fix the regular language L accepted by a 'permutation' DFA (deterministic finite automaton) that has $\{0,1\}$ as the set of input symbols, $\{q_0, q_1, q_2\}$ as the set of states (with q_0 as the start state and q_2 as the accepting state) and δ as the transition function specified by $\delta(q_i, 1) = q_{i+1 \mod 3}, \delta(q_0, 0) = q_1, \delta(q_1, 0) = q_0, \delta(q_2, 0) = q_2$.

Consider a function $h(x_1, \ldots, x_m)$ (where each x_j is a bit, where $j \in \{1, \ldots, m\}$) that outputs 1 if $x \in L$, and 0 otherwise. Assume each x_j is assigned to some party and each party is assigned at most $k = o(n)$ bits at fixed (not necessarily consecutive) positions. To evaluate this function, one can devise a simple non-abelian program $h(x_1, \ldots, x_m) = f(\pi_{1,x_{i_1}} \cdot \ldots \cdot \pi_{t,x_{i_t}})$, where all π_{t,x_t}'s are in the group S_3 (where S_3 denotes a permutation group, whose elements are permutations of a set $M = \{1,2,3\}$, and the group operation is the composition of permutations), and for each t, $\pi_{t,0} = (1,2)(3)$ and $\pi_{t,1} = (1,2,3)$ [10]. The function $f(\pi)$ outputs 1 if and only if $\pi(1) = 3$ [11]. Note that this non-abelian program is over a 'small' group. However, it is not clear how to devise an abelian program that works with 'small' groups of similar size for this function.

We point that it is always the case that a function $h : D^m \to \{0,1\}$ can be expressed as an abelian program that works over the group $\mathbb{Z}^m_{|D|}$ – Each party P_i simply computes $x_i \vec{e}_i$, where $\vec{e}_i$ is the i'th vector

[9] Note that even if the IFSS computes a 'masked' output (like in the case of a non-root gate in the construction for tree-like formulas), this would still violate security as the adversary could learn additional information about private inputs of honest parties involved in this gate just by comparing these multiple masked outputs corresponding to different common inputs.

[10] We use the cycle notation to express permutations. E.g. $\pi = (12)(3)$ denotes the permutation where $\pi(1) = 2, \pi(2) = 1$ and $\pi(3) = 3$ as $(1,2)$ denotes the cyclic permutation and 3 is left unchanged.

[11] For e.g. consider the $x = x_1, \ldots, x_6 = 001011$, where each P_i ($i \in \{1,2,3\}$) holds x_i and x_{i+3}. One can check that $x \in L$ and $g(\pi_{1,0} \cdot \pi_{2,0} \cdot \pi_{3,1} \cdot \pi_{4,0} \cdot \pi_{5,1} \cdot \pi_{6,1}) = 1$.

in the standard basis, which can be aggregated to compute the sum, denoting the entire input. The problem is that this group is too large, and it is not clear (to us) how to do much better with abelian programs for the above DFA example.

Non-Abelian Programs versus 'Tree'-based Formulas. In terms of feasibility (within the $o(n)$ domain), the formula-based construction is more expressive, within our $\mathcal{BC}$ constrains. This holds since we can simulate a non-abelian program involving $k \cdot n$ terms via a (nearly) balanced tree of depth $d = \log(k) + \log(n)$, using associativity of multiplication in the group. So, a read-k program would result in a read-k formula. Plugging in our constructions using the two approaches would result in their $\mathcal{BC}$ being very close, with only polylogarithmic overhead for the formula-based construction.

Next, we discuss whether there also exists a transformation in the other direction i.e. from formula to non-abelian programs. Given a generalized read-k formula, it is not always clear how to devise a (non)-abelian program with small overhead as above. In particular, the generic transformation due to Barrington [Bar86] transforming a formula into a BP results in a BP of length in quadratic in formula *size*, and constant width, which is already $\Omega(n^2)$ for non-trivial formulas (with size at least n), and is prohibitively expensive for $\mathcal{BC}$. In fact, the resulting BP is already a permutation BP, but this does not help us, due to the large k (size of formula $\Omega(n^2)$ implies that k must be $\Omega(n)$). Despite the fact that non-abelian programs allow computing an arbitrarily complicated function f after a sequence of compositions on non-abelian group elements (where the sequence can be visualized as a permutation BP), it is not clear how this would help in the above transformation.

Next, we observe that the formula-based solution works for branching programs [12] (BP's). This is because the formulas with the 'generalized gates' can support arbitrary transformations induced by inputs between the BP layers, which can be composed due to associativity of the function. As long as the width and parameter k of a 'k-read BP' is 'small', the formula-based approach would be $\mathcal{BC}$-efficient.

The above raises a question regarding if non-abelian programs support BPs. We observe that they would support a special kind of BPs, namely, permutation BPs [13]. In such a program of width w, the transition from root forward can be viewed as a composition of permutations in S_w, which defines a non-abelian program. The output is then determined based on whether the composed permutation maps 1 to an accept node (where the first and last transition are adapted to be permutations in a natural way). If $w = n$, the resulting group elements are too large for sublinear $\mathcal{BC}$ (even when transferring a single element). If all permutations actually fall in a subgroup of S_w, we could work in that subgroup and hope to obtain efficiency. Notably, non-abelian programs would not support general branching BP, as it may not be possible to map the transformations between layers in a general BP to a group structure (as it may not have an inverse or identity element). Lastly, we point that permutation BPs are somewhat restricted, and moving from (regular) BP to a permutation BP may have large costs in terms of w.

The above discussion argues that formulas are generally more expressive. However, there are still situations where using the non-abelian program approach is more useful. For instance, consider functions for which the resulting $\mathcal{BC}$ for the non-abelian program based protocol is constant (as in the permutation DFA example above). In such a case, moving from the non-abelian program construction to the formula-based construction, incurs a super-constant overhead. In particular, a $O(k \cdot d)/O(k) = \log(n)$ overhead for the information-theoretic construction is incurred. So, the former construction would still be preferred if one wishes to achieve the 'ideal' best possible notion of $\mathcal{BC}$-efficiency, namely constant $\mathcal{BC}$.

[12] A directed acyclic graph in which the nodes are labeled by input variables and every nonterminal node has two outgoing edges, labeled by 0 and 1

[13] In a nutshell, these are layered branching programs, where every level's transitions, for each input value $x_i = b$ constitute a permutation $\pi_{i,b}$ [Bar85]. Another difference between it and standard BP's is the way acceptance is defined. There is no root and accept/reject nodes, but rather a single resulting composed permutations, and acceptance/rejection is defined by belonging to one of two sets of output permutations, partitioning S_w. The width of such a program is the number of nodes, w, in each layer except for the first and last ones.

3 Preliminaries

Notation. The cryptographic security parameter will be denoted by λ. The n parties $\{P_1, \ldots, P_n\}$ are pair-wise connected by secure and authentic channels, where n is polynomially bounded. We operate with semi-honest security and assume that any adversary can passively corrupt up to $n-1$ parties.

We evaluate functions $f : \mathcal{X} \to \mathcal{Y}$ from a function class $\mathcal{F}$. We will often assume that $\mathcal{X}$ and $\mathcal{Y}$ are groups endowed with an operation. We consider both abelian and non-abelian groups.

3.1 Security Model

We prove the security of our protocols based on the standard real/ideal world paradigm. 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:*

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

3.2 Definitions

Definition 2 (Bottleneck Complexity of a Protocol). *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 $\mathcal{BC}(\Pi) = \max_{i\in[n]} \mathcal{CC}_i$.*

We use the formal definition of [BJPY18] for bottleneck complexity. Informally, the bottleneck complexity of a protocol is the maximum communication complexity required by any party in the protocol execution. We consider a protocol Π to be $\mathcal{BC}$-efficient if the $\mathcal{BC}$ is sublinear in the number of total parties.

Definition 3 (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.*

Definition 4 (Non-Abelian Programs). *Let $(G, \cdot)$ be a non-abelian group, $x_1, \ldots, x_n$ be inputs from domain D, and $\mathcal{H}^G_D$ be the set of functions $h : \{D\}^n \to \{0,1\}$ of the form $h(x_1, \ldots, x_n) = f(\pi_{1,x_{i_1}} \cdot \ldots \cdot \pi_{t,x_{i_t}})$, where each $\pi_{j,x_{i_j}}$ is an element of a group G that depends only on j, x_{i_j} (where $i_j \in \{1, \ldots, n\}$) for some $f : G \to \{0,1\}$. We call such functions h non-abelian programs.*

3.3 Primitives

Garbled Circuits. 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 $\mathbf{K}[x] = (K_1^{x_1}, \ldots, K_\ell^{x_\ell})$.

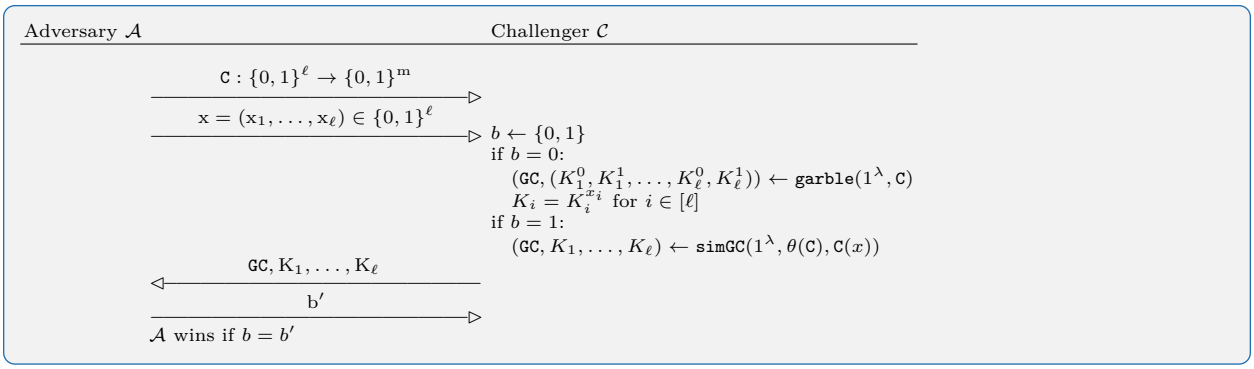
Next, we formally define the security notions we require for a garbling scheme. When garbled circuits are used in such a way that decoding information is used separately, *obliviousness* requires that a garbled circuit together with a set of labels reveals nothing about the input the labels correspond 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 `GC` satisfies *privacy* if there exists a simulator `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:



4 Incremental Function Secret-Sharing

We begin by defining *incremental function secret-sharing (*IFSS*)*, which allows a set of parties to evaluate a hidden function on a joint input. Diverging from the original definition of function secret sharing [BGI16], IFSS requires shares to be aggregated incrementally on a chain, in a $\mathcal{BC}$-efficient communication pattern. IFSS can be instantiated with garbled circuits or one-time truth tables and can be used in $\mathcal{BC}$-efficient protocols for particular function classes.

Definition 6 (Incremental Function Secret-Sharing). *An n-party incremental function secret-sharing (IFSS) scheme for a function class $\mathcal{F}$ is a pair of PPT algorithms $(\mathtt{Gen}, \mathtt{Eval})$ with the following syntax:*

- $\mathtt{Gen}(1^\lambda, f)$*: On input security parameter 1^λ and function description $f \in \mathcal{F}$,* $\mathtt{Gen}$ *outputs keys $(k_1, \ldots, k_n)$;*
- $\mathtt{Eval}(i, k_i, x, y_{i+1})$*: On input party index i, a key k_i, input string x, and the output of the next party y_{i+1}, the algorithm* $\mathtt{Eval}$ *outputs a value y_i;*

We require the following:

Correctness: For all $(f : \mathcal{X} \to \mathcal{Y}) \in \mathcal{F}, x \in \mathcal{X}$ we require:

$$\mathtt{Eval}(1, k_1, x, \mathtt{Eval}(2, k_2, x, \ldots, \mathtt{Eval}(n, k_n, x, \bot))) = f(x)$$

except with negligible probability.

Privacy: Let $\mathcal{H}$ be the set of honest parties. Then if $\vec{k} \leftarrow \mathtt{Gen}(1^\lambda, f)$, we define $\vec{k}_{-\mathcal{H}}$ to be $\vec{k}$ where we replace, for all $i \in \mathcal{H}$, k_i with $\bot$. We also define $\mathtt{Eval}^{\mathcal{H}}(\vec{k}, x)$ to compute, for $i = n, \ldots, 1$, $y_i = \mathtt{Eval}(i, k_i, x, y_{i+1})$ (with $y_{n+1} = \bot$), and then output y_i for all $i \in \mathcal{H}$.

We say that an IFSS satisfies privacy if, there exists a PPT simulator $\mathtt{Sim}$ such that for all $f \in \mathcal{F}, \mathcal{H} \subset [n], x \in \mathcal{X}$:

$$\{k_{-\mathcal{H}}, \mathtt{Eval}^{\mathcal{H}}(\vec{k}, x) : \vec{k} \leftarrow \mathtt{Gen}(1^\lambda, f)\}_{\lambda, f, x}, \text{ and } \{\mathtt{Sim}(1^\lambda, \mathcal{H}, x, f(x))\}_{\lambda, f, x}$$

are (unconditionally or computationally) indistinguishable.

Bottleneck Complexity: We define the bottleneck complexity $\mathcal{BC}$ of an IFSS for $\mathcal{F}$ as the expected size of the largest y_i, for all $i \in [n], f \in \mathcal{F}, x \in \mathcal{X}$.

4.1 Instantiating IFSS

We show two instantiations of IFSS, one based on one-way functions and one with unconditional security.

With Unconditional Security. IFSS can be implemented with information theoretic security using an approach similar to the OTTT protocol [IKM+13] (as observed in [BGI16]). The construction is as follows: $\texttt{Gen}(1^\lambda, f)$ chooses random vectors $T_1, \ldots, T_n$ whose dimensionality is $|\mathcal{X}_f|$, the size of the input domain of f, such that for all possible inputs $x \in \mathcal{X}_f$, $\sum_i T_i[x] = f(x)$. `Gen` then outputs $k_i = T_i$. The evaluation algorithm $\texttt{Eval}(i, k_i, x, y_{i+1})$ outputs $y_i = T_i[x] + y_{i+1}$ (for $y_{i+1} \neq \perp$ and $y_i = T_i[x]$ otherwise).

The protocol satisfies correctness since by construction $y_1 = \sum_i T_i[x] = f(x)$. It also satisfies unconditional privacy: The simulator $\texttt{Sim}(1^\lambda, \mathcal{H}, x, f(x))$ samples $k_{-\mathcal{H}} = \{T_i\}_{i \notin \mathcal{H}}$ as a set of uniform random strings of length $|\mathcal{X}_f|$, and random $\{z_i\}_{i \in \mathcal{H}}$ from $\mathcal{Y}_f$. Then it simulates the outputs of the `Eval` function as follows: it sets $y_1 = f(x)$, and $y_{i+1} = y_i - ((i \in \mathcal{H})\ ?\ z_i : T_i[x])$ [14] for all $i < n$, and finally outputs $(k_{-\mathcal{H}}, \{y_i\}_{i \in \mathcal{H}})$. Indistinguishability follows since in the simulation, like in the real protocol, the corrupt parties receive uniformly random k_i, and the y_i values are uniformly distributed under the constraint that y_1 is the result of the computation.

Note that since the constructions leaks $\mathcal{X}, \mathcal{Y}$, we assume that for all $f \in \mathcal{F}$ $\mathcal{X}_f = \mathcal{X}, \mathcal{Y}_f = \mathcal{Y}$. For this IFSS, $\mathcal{BC} = \mathcal{O}(\log |\mathcal{Y}|)$. (Note that the size of the keys can be exponential in the input size, namely $\mathcal{O}(\log |\mathcal{Y}| \cdot |\mathcal{X}|)$, like the original OTTT protocol).

From One-Way Functions. IFSS can be implemented from garbled circuits (which in turn can be implemented from one-way functions) by abstracting the "Phase 2" step of the protocol for abelian programs presented in [ORS22]. The construction is as follows: the algorithm $\texttt{Gen}(1^\lambda, f)$ runs $(\texttt{GC}, \mathbf{K}) \leftarrow \texttt{garble}(1^\lambda, f)$. Then it picks uniformly random $\{\mathbf{K}_i\}_{i \in [n]}$ under the constraint that $\sum_i \mathbf{K}_i = \mathbf{K}$. Finally it outputs $k_i = \mathbf{K}_i$ for all $1 \neq i \in [n]$ and $k_1 = (\texttt{GC}, \mathbf{K}_1)$. The evaluation algorithm $\texttt{Eval}(i, k_i, x, y_{i+1})$, for all $i \neq 1$, selects the shares of the encoding information of $k_i = \mathbf{K}_i$ that correspond to x i.e., $\mathbf{K}_i[x] = ((K_i)_1^{x_1}, \ldots, (K_i)_\ell^{x_\ell})$ where $\ell = \lceil \log |\mathcal{X}| \rceil$, and finally outputs $y_i = \mathbf{K}_i[x] + y_{i+1}$ (for $y_{i+1} \neq \perp$ and $y_i = \mathbf{K}_i[x]$ otherwise).

For $i = 1$ the `Eval` algorithm follows the instructions above to produce y_1, and finally outputs $\texttt{eval}(\texttt{GC}, y_1)$.

The protocol satisfies correctness since by construction $y_1 = \sum_i \mathbf{K}_i[x] = \mathbf{K}(x)$, and by correctness of the garbling scheme $\texttt{eval}(\texttt{GC}, \mathbf{K}[x]) = f(x)$ except with negligible probability.

It also satisfies computational privacy: The simulator $\texttt{Sim}(1^\lambda, \mathcal{H}, x, f(x))$ runs the simulator for the garbled circuits $(\texttt{GC}, \mathbf{Y}) \leftarrow \texttt{simGC}(1^\lambda, \theta(\mathcal{F}), f(x))$, where $\mathbf{Y}$ is the set of ℓ labels that make the simulated garbled circuit `GC` output $f(x)$. The simulator then picks $\{\mathbf{K}_i\}_{i \notin \mathcal{H}}$, a set of uniform random strings of the same length as $\mathbf{K}$, and random strings $\{z_i\}_{i \in \mathcal{H}}$ of the same length as $\mathbf{K}[x]$. Then it simulates the outputs of the `Eval` function as follows: it sets $y_1 = \mathbf{Y}$ and $y_{i+1} = y_i - ((i \in \mathcal{H})\ ?\ z_i : \mathbf{K}_i[x])$ for all $i < n$, and finally outputs $(k_{-\mathcal{H}}, \{y_i\}_{i \in \mathcal{H}})$.

An adversary $\mathcal{A}$ that can distinguish between the real and simulated distribution can easily be used to break the privacy property of the underlying garbling scheme. The reduction $\mathcal{B}$ queries the GC challenger $\mathcal{C}$ on input f, x and receives $(\texttt{GC}, \mathbf{Y})$ in return. It then picks random $\mathbf{K}_i$ for all $i \notin [n]$ and computes the y_i as the simulator described above. The resulting distribution corresponds to the real protocol execution if the GC challenger sampled $b = 0$, or the simulated one if $b = 1$, therefore the reduction $\mathcal{B}$ wins in the GC privacy game with the same advantage as the IFSS adversary $\mathcal{A}$ distingushes between the real and simulated view.

Note that privacy of the GC scheme leaks some information $\theta(f)$ about the function f, therefore we assume for simplicity that for all $f \in \mathcal{F}$, $\theta(f) = \theta(\mathcal{F})$. However, given an upper bound of the size of $f \in \mathcal{F}$ it is possible to remove this requirement using universal circuits, albeit with an efficiency loss, and this is reflected in the Lemma below. For the GC based IFSS, $\mathcal{BC} = \mathcal{O}(\lambda \cdot \log |\mathcal{X}|)$. Note that the size of the keys is polynomial in the input size for this instantiation, namely $\mathcal{O}(\lambda \cdot (\log |\mathcal{X}| + |f|))$ for the first party and $\mathcal{O}(\lambda \cdot \log |\mathcal{X}|)$ for the others.

The discussion in this subsection can be summarized in the following:

Lemma 1. *Let $\mathcal{F}$ be a class of functions $f : \mathcal{X} \to \mathcal{Y}$, λ the security parameter. Then for all n:*

- *It is possible to implement* IFSS *for $\mathcal{F}$ with $\mathcal{BC} = \mathcal{O}(\log |\mathcal{Y}|)$ with unconditional security.*
- *If one-way functions exist, it is possible to implement* IFSS *for $\mathcal{F}$ with $\mathcal{BC} = \mathcal{O}(\lambda \cdot \log |\mathcal{X}|)$.*

[14] Here, $a\ ?\ b : c$ is used to denote the folllowing – if a holds, then b; else c.

In particular, the $\mathcal{BC}$ complexity in both cases is independent of n. Note however that the size of the correlated randomness in the variant with unconditional security is exponential in the input size.

Using IFSS *in the compiler of [BJPY18].* The work of [BJPY18] presents a compiler that transforms an insecure protocol to secure protocol while preserving $\mathcal{BC}$. The compiler is based on the tool of 'incremental FHE', which is similar to FHE except that its 'joint' public key and decryption of ciphertext can be computed by incrementally combining shares provided by different parties. The main idea of the compiler is to execute the insecure protocol under the hood of (incremental) FHE to compute the encryption of the output (say ciphertext ct). Next, the parties combine their partial decryptions (computed locally by each party using its share of secret key) corresponding to ct in an incremental manner to reconstruct the final decrypted output.

We analyze whether this compiler can be viewed in terms of the common two-step approach of our constructions (elaborated in the technical overview, where the first step is to compute a 'masked' aggregate common input and the second step is to use IFSS to evaluate a hidden function on this input). Recall that in our constructions, the first step involves masking the inputs using random values from setup (and either aggregating them by incremental computation or concatenating them) and the second step involves using IFSS to carry out the unmasking and compute the relevant function. On the other hand, in the above compiler, the first step uses FHE to compute the encryption of output ct directly. We observe that considering ct to be the common joint input, now IFSS can in fact be used to carry out the 'decryption' function of FHE. In some more detail, IFSS could be used to evaluate the 'hidden' function which has the secret decryption keys hardcoded and computes the decryption of the ciphertext ct. The above approach would result in making the compiler rely on correlated randomness, but would allow to instantiate it using any (non-incremental) FHE scheme. This shows that IFSS can serve as a general useful building block in $\mathcal{BC}$-efficient constructions.

5 Low $\mathcal{BC}$-complexity for Abelian Programs from IFSS

In this section, we generalize the results of [ORS22] using the IFSS primitive defined above. Doing so allows us to achieve an information-theoretic $\mathcal{BC}$-efficient protocol for abelian programs.

Note that we can't use IFSS directly in MPC protocols for two reasons: first, all parties in IFSS would need to know all the inputs $\vec{X}$. This could be fixed by introducing a mask $\vec{R}$, reveal $\vec{X}+\vec{R}$ to all parties, and then modify the function so that it removes the mask securely inside the IFSS. The second issue is that revealing (even a potentially masked) $\vec{X}$ to all parties would lead to high $\mathcal{BC}$ since $\mathcal{O}(|\vec{X}|) = \mathcal{O}(n)$.

Recall that an abelian program h can be expressed as $h(\vec{X}) = f(\sum_{i=1}^{n} X_i)$ for some $f : G \to \{0,1\}$, where G denotes an abelian group. We observe that the specific function class of abelian programs has a special structure that allows us to view it as a single-input function rather than an n-input function h. Exploiting this observation, we fix the above problems as follows: first, the trusted dealer picks a (single) random $R \in G$, defines the function $f'_R(Z) = f(Z - R)$, and then gives all parties P_i an additive share r_i of R, which they can use to mask their inputs, and an IFSS key k_i for the function f'_R.

In the protocol, the parties securely compute a masked sum of their inputs (say $Z = X + R$, where R denotes the mask and X denotes the sum of inputs) in a $\mathcal{BC}$-efficient way over a chain (similar to the protocol of [ORS22]). The parties mask their inputs X_i with r_i as $Z_i = X_i + \mathsf{r}_i$, add it to $\sum_{j=1}^{i-1} Z_j$, which they received from the previous party P_{i-1}, and send the result to P_{i+1}. The last party in the chain then can recover $Z = X + R$, and begins the second phase where each party P_i sends Z together with $y_i = \mathtt{Eval}(i, k_i, Z, y_{i+1})$ to P_{i-1}, so that P_1 can finally retrieve $f'_R(Z) = f(Z - R) = f(X)$. Intuitively, security follows from the use of the masks and privacy of IFSS which hides which function (and therefore mask) was used. The $\mathcal{BC}$ of the protocol is inherited from the IFSS plus the size of elements in G, both independent of n.

Figure 5.1: $\Pi_{\mathtt{abl}}$

Private input. Each party P_i has input $X_i \in G$ from group G.
Output. $y = f(\sum_{i=1}^{n} X_i)$, where the output is a single bit.
Correlated Randomness Setup. The setup involves the following:

1. For each $i \in [n]$, sample $\mathsf{r}_i \in G$, such that $\sum_{i=1}^{n} \mathsf{r}_i = R$.
2. Define the function $f'_R(Z)$ that computes $f(Z-R)$ on the input Z. Use an IFSS scheme to compute $(k_1, \ldots, k_n) \leftarrow \mathtt{Gen}(1^\lambda, f'_R)$.
3. Output (r_i, k_i) to P_i for each $i \in [n]$.

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

Phase 1 (Round 1 to Round n). *(Input Masking)* In round i, P_i does the following:
- If $i = 1$, let $V_i = \mathsf{r}_i + X_i$.
- If $i \neq 1$, let V_{i-1} denote the message received from P_{i-1} during the previous round. Compute $V_i \leftarrow V_{i-1} + X_i + \mathsf{r}_i$.
- If $i < n$, send V_i to P_{i+1}.
- If $i = n$, set $Z = V_n$.

Phase 2 (Round $n+1$ to Round $2n$) *(IFSS Evaluation)* Each P_i does the following in sequence, starting from $i = n$ to 1:
- If $i = n$, set $y_i = \mathtt{Eval}(i, k_i, Z, \perp)$.
- If $i \neq n$, parse the message received from P_{i+1} in the previous round as (Z, y_{i+1}). Compute $y_i = \mathtt{Eval}(i, k_i, Z, y_{i+1})$.
- If $i \neq 1$ Send (Z, y_i) to P_{i-1}.

Output Computation. P_1 sets the output $y = y_1$.

Phase 3 (Round $2n+1$ to $3n$) *(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.

Theorem 1. *Protocol $\Pi_{\mathtt{abl}}$ securely computes the abelian program h against a semi-honest adversary corrupting upto $n-1$ parties. The $\mathcal{BC}$ of $\Pi_{\mathtt{abl}}$ is $O(\log |G|)$ and $O(\lambda \log |G|)$ for the information-theoretic and the computational variant respectively.*

Proof. Let $\mathcal{I}$ and $\mathcal{H} = \mathcal{P} \backslash \mathcal{I}$ denote the set of indices corresponding to corrupt and honest parties respectively. Since we are running a protocol on a chain, it is useful to be able to talk about corrupt parties who receive messages from honest parties, and we therefore define $\mathcal{I}_L$ (resp. $\mathcal{I}_R$) to be the sets of all $i \in \mathcal{I}$ such that $i - 1 \in \mathcal{H}$ (resp. $i + 1 \in \mathcal{H}$).

To prove security, we define a simulator $\mathcal{S}$ that simulates the real-world view of the corrupt parties. Recall that $\mathcal{S}$ is given $(\mathcal{I}, \{x_i\}_{i \in \mathcal{I}}, y)$.

Setup simulation. Run $(\{k_i\}_{i \in \mathcal{I}}, \{y_i\}_{i \in \mathcal{H}}) \leftarrow \mathsf{simIFSS}(1^\lambda, \mathcal{H}, Z', y)$, where $\mathsf{simIFSS}$ denotes the simulator of the IFSS scheme's $\mathtt{Gen}$ and $\mathtt{Eval}$ functionality for a function class $\mathcal{F}$ computing $f'_R(Z) = f(Z-R)$ (note that the function class is independent of the value R in the function). Z' is chosen uniformly at random from the elements of G.

Additionally, for each $i \in \mathcal{I}$, sample r_i uniformly in G, and include (k_i, r_i) in the view of P_i.

Phase 1 Simulation. We need to simulate V_{i-1} for all $i \in \mathcal{I}_L$. We do so by choosing uniformly random V_{i-1} from G for all such $i \in \mathcal{I}_L$, except the largest one, which we denote by $\tilde{i}$, which we simulate by computing $V_{\tilde{i}-1} = Z' - \sum_{j \geq \tilde{i}} (X_j + \mathsf{r}_j)$ (In other words we define the message sent by the honest party with the largest index, to be consistent with the Z' which was chosen when simulating the IFSS, the input of the corrupt parties and their shares of R which were already defined during setup).

Phase 2 Simulation. We include in the view of all P_i with $i \in \mathcal{I}_R$ the tuple (Z', y_{i+1}), where y_{i+1} was received from the IFSS simulator.

Phase 3 Simulation. We include in the view of all P_i with $i \in \mathcal{I}_L$ the result y.

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 the values k_i for all $i \in \mathcal{I}$ and all y_{i+1} for $i \in \mathcal{I}_R$ are computed using the IFSS simulator on input $(\mathcal{H}, Z, f(x))$.

  This is in contrast to the previous hybrid, where the true IFSS evaluation is used instead of a simulator, changing the k_i of corrupt parties and y_i of honest parties. Indistinguishability follows from the privacy of IFSS.
- Hyb_2 : Same as Hyb_1, except that R is not used anymore to define the $\mathsf{r}_i : i \in \mathcal{I}$, which are instead just chosen at random from G. Since the r_i of the honest parties are not part of the view the two distributions are identically distributed.
- Hyb_3 : Same as Hyb_2, except that a random Z' is input to the IFSS simulator, and $V_i : i \in \mathcal{I}$ are simulated as decribed in "Phase 1 Simulation". This is in contrast to the previous hybrid, where Z is computed from the V_i values, and V_i are computed based on the parties' inputs. Since the r_i of the honest parties are not part of the view the two distributions are identically distributed. Note that in this hybrid we do not use the inputs of the honest parties anymore.

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

$\mathcal{BC}$-Analysis. We note that in Phase 1 and 3, the maximum communication complexity incurred by a party is $\log|G|$. In Phase 2, a party incurs the $\mathcal{BC}$ of the IFSS instance (in addition to $|Z| = \log|G|$), which is $O(1)$ for the information-theoretic instantiation and $O(\lambda \log|G|)$ for the computational instantiation. We can thus conclude that the resulting $\mathcal{BC}$ of the information-theoretic protocol for abelian programs is $O(\log|G|)$. The computational variant (which is the same as the construction in [ORS22]) has a $\mathcal{BC}$ of $O(\lambda \log|G|)$.

5.1 $\mathcal{BC}$-efficient MPC for maximum

We introduce several protocols, which directly use $\Pi_{\mathtt{abl}}$ as a subprotocol in a $\mathcal{BC}$-efficient way.

The first protocol finds the maximum among the set of private inputs. Consider a set of parties $\{P_1, \ldots, P_n\}$ where each party P_i holds an element $X_i \in [\mathsf{S}]$. Following is a protocol to compute $y = \max(X_1, X_2, \ldots, X_n)$. We use the $\Pi_{\mathtt{abl}}$ protocol from Figure 5.1 as a subprotocol and run this subprotocol S times, once for each possible value of the maximum. (A more efficient version is discussed later). Each abelian program outputs 1 if at least one party's input is greater than that possible maximum value. The maximum is therefore the largest element for which the execution of $\Pi_{\mathtt{abl}}$ outputs 1.

More specifically, each abelian program fixes one possible output value from the set and takes a single-bit input from each party. Before each run of the protocol, the parties evaluate a function locally with input bit 0 if their input is less than the value corresponding to that abelian program and 1 otherwise. The abelian program takes the sum of these bit inputs, and outputs 1 if the sum is nonzero. We can consider the abelian group $\mathbb{Z}_{n+1}$ (integers modulo $(n+1)$) for this purpose. Therefore, each abelian program associated with a value less than or equal to the maximum of the set outputs 1. Note that each subprotocol being run essentially evaluates the OR function between all parties' single-bit inputs. To fit an abelian program structure, each program adds the parties' input bits and compares the sum to 0. Other $\mathcal{BC}$-efficient specialized protocols for evaluating OR could also be used here, such as the one proposed originally as a warm-up by [ORS22]. The formal description of this max protocol appears in Figure 5.2.

Figure 5.2: $\Pi_{\max}$

Private input. Each party P_i has input X_i from set $[\mathtt{S}]$.
Setup. Run the setup phase of $\Pi_{\mathtt{abl}}$.
Output. $y = \max(X_1, X_2, \ldots, X_n)$
The Protocol. The following steps are run in the online phase of the protocol:

- At the beginning of the protocol, each party i computes $x_i^j = X_i \geq j$ for $j = 1, 2, \ldots, \mathtt{S}$.
- Then run the online phase of $\Pi_{\mathtt{abl}}$ with inputs $(x_1^j, x_2^j, \ldots, x_n^j)$ and a function that checks if the sum of inputs is greater than 0 and outputs 1 if so, $[\mathtt{S}]$ times in parallel for $j \in [\mathtt{S}]$, receiving outputs $Y = y_1, y_2, \ldots, y_{\mathtt{S}}$.
- Output $\max(X_1, X_2, \ldots, X_{\mathtt{S}}) = \text{argmax}_i(y_i \in Y : y_i = 1)$.

Correctness. For correctness, notice that the final output of the protocol is the largest value (say j) for which $\Pi_{\mathtt{abl}}$ returns 1, implying that there exists at least one party with a value equal to j and no party with a value greater than j. Therefore j is the maximum.

Privacy. Privacy of the protocol follows from the security of $\Pi_{\mathtt{abl}}$ and the fact that the intermediate outputs $Y = y_1, y_2, \ldots, y_{\mathtt{S}}$ can be efficiently simulated given the final output y, since Y is a vector of bits such that $y_i = 1$ if $i \leq y$.

$\mathcal{BC}$-Analysis. Each run of $\Pi_{\mathtt{abl}}$ has $\mathcal{BC}$ of $\mathcal{O}(\lambda \log n)$ for the GC implementation and $\mathcal{O}(\log n)$ for the OTTT implementation. Since $\Pi_{\mathtt{abl}}$ is run $\mathtt{S}$ times, the total $\mathcal{BC}$ of the protocol is $\mathcal{O}(\lambda \mathtt{S} \log n)$ or $\mathcal{O}(\mathtt{S} \log n)$.

Alternatively to running protocol $\Pi_{\mathtt{abl}}$ $\mathtt{S}$ times in parallel, we could also use a binary search approach and run the protocol. In that case, we would run $\Pi_{\mathtt{abl}}$ with $j = \mathtt{S}/2$. If the result is 1, run the protocol with $j = 3\mathtt{S}/4$. If the result is 0, run the protocol with $j = \mathtt{S}/4$, narrowing the options for the true maximum by half each time. This requires $\Pi_{\mathtt{abl}}$ to be run $\log \mathtt{S}$ times, resulting in a $\mathcal{BC}$ of $\mathcal{O}(\lambda \log^2 \mathtt{S})$ or $\mathcal{O}(\log^2 \mathtt{S})$. Since this protocol using binary search does not allow the runs of $\Pi_{\mathtt{abl}}$ to be done in parallel, the number of rounds necessary for this protocol is $\log \mathtt{S}$ times greater; this results in a round complexity of $3n$ for the parallelized approach and $3n \log \mathtt{S}$ for the binary search approach.

Function of Maximum. Next, we present a $\mathcal{BC}$-efficient MPC protocol to compute a function g of the maximum of elements from $[\mathtt{S}]$. Since we are computing a function of the maximum, we cannot leak the maximum itself; therefore, we do not run $\mathtt{S}$ copies of $\Pi_{\mathtt{abl}}$ in parallel, but a single one. The idea is the following: like in the previous protocol, each party P_i converts their input $X_i \in [\mathtt{S}]$ into a bit-vector $\vec{x}_i$ of size $\mathtt{S}$, such that the first X_i entries are 1 and the remaining ones are 0. Then, use a single instance of $\Pi_{\mathtt{abl}}$ computing the function:

$$f_{\mathsf{max}}(\vec{x}_1, \ldots, \vec{x}_n) = g\left(\text{argmax}_j\left(z_j = \left(\sum_{i \in [n]} \vec{x}_i[j]\right) : z_j > 0\right)\right)$$

Figure 5.3: $\Pi_{f(\max)}$

Private input. Each party P_i has input X_i from $[\mathtt{S}]$.
Output. $y = f(\max(X_1, X_2, \ldots, X_n))$
Correlated Randomness Setup. Run the setup for $\Pi_{\mathtt{abl}}$ for the function f_{max} defined above.
The Protocol. Each party P_i defines a vector $\vec{x}_i$ such that $\vec{x}_i[j] = 1$ if $X_i \geq j$ or 0 otherwise. Then the parties run the online phase of the $\Pi_{\mathtt{abl}}$ protocol for the function f_{max}.

The idea is to compute the sum of the bit vectors (i.e. the sum of the $\vec{x}_i$'s, where the sum is computed over the abelian group $\mathbb{Z}_{n+1}$ for each bit position), then identify the maximum bit position (among 1 to $\mathtt{S}$) to compute the maximum, upon which the function g can now be computed. Correctness and privacy follow easily from the properties of $\Pi_{\mathtt{abl}}$.

$\mathcal{BC}$-Analysis. The $\mathcal{BC}$ of $\Pi_{f(\max)}$ is simply the $\mathcal{BC}$ of the abelian protocol instance used which is $\mathcal{O}(\lambda \mathtt{S} \log n)$ for the GC implementation and $\mathcal{O}(\mathtt{S} \log n)$ for the OTTT implementation.

6 $\mathcal{BC}$-efficient MPC for product of Non-Abelian Group elements

We build on protocol $\Pi_{\mathtt{abl}}$ for abelian programs (which can be expressed as a function on sum of inputs) to design $\mathcal{BC}$-efficient MPC protocol for non-abelian programs (which can be expressed as a function of a non-abelian product [15]). To offer an example, permutation branching programs can be evaluated with this approach. The main difference between abelian and non-abelian programs is that for non-abelian programs, the operations on the input values and masking values must be done in a specific order (determined by the input sequence forming the non-abelian product). This order can easily be taken into account during computation, masking, and unmasking, by computing a product, where party P_i with input X_i can receive r_i such that $\prod_{i\in[n]} \mathsf{r}_i = R$ and a masked result is computed along a chain as $\mathsf{r}_n \cdot \big(\mathsf{r}_{n-1} \cdot \cdots \cdot (\mathsf{r}_2 \cdot (\mathsf{r}_1 \cdot X_1) \cdot X_2) \cdot \ldots X_{n-1}\big) \cdot X_n = R \cdot X_1 \cdot X_2 \cdot \ldots X_{n-1} \cdot X_n$. The inverse R^{-1} can be used to unmask, as $R^{-1} \cdot R \cdot X_1 \cdot X_2 \cdot \ldots X_{n-1} \cdot X_n$.

Furthermore, we consider read-k non-abelian programs. Unlike the case of abelian programs, if a party's input appears twice in the non-abelian product, this cannot be locally combined by this party (as the input might appear in non-consecutive positions and the non-abelian group need not be commutative). In our protocol, we define a sequence of parties, each of whom have a single input from some domain. We define a mapping from a party's input value to an element of a non-abelian group, which can be chosen dependent on the position in the input sequence. This dependency allows different group elements to be chosen for a single input value at different positions in the non-abelian program.

More formally, consider a group $(G, \cdot)$ and a set of parties $\{P_1, \ldots, P_n\}$ where each party P_i holds an input from some domain $X_i \in D$. Following is a protocol to compute $y = f(\pi_{1,X_{i_1}} \cdot \pi_{2,X_{i_2}} \cdot \ldots \cdot \pi_{t,X_{i_t}})$, based on an order of X_i values fixed for the function to yield $X_{i_1}, \ldots, X_{i_t}$ and a mapping from an index $\alpha \in [t]$ and input X to group element $\pi_{\alpha,X}$.

Figure 6.1: $\Pi_{\mathtt{nonabl}}$

Private input. Each party P_j for $j \in [n]$ has input $X_j \in D$ from some domain D, and there exists a public sequence of indices $i_1, \ldots, i_t$ with $i_\alpha \in [n]$ for all $\alpha \in [t]$. There also exists a public input mapping that outputs a group element $\pi_{\alpha,x}$, given an index $\alpha \in [t]$ and an input $x \in D$.
Output. $y = f(\pi_{1,X_{i_1}} \cdot \pi_{2,X_{i_2}} \cdot \ldots \cdot \pi_{t,X_{i_t}})$
Correlated Randomness Setup. The setup involves the following:

1. For each $\alpha \in [t]$, sample $\mathsf{r}_\alpha \in G$, such that $\prod_{\alpha=1}^{t} \mathsf{r}_\alpha = R$.
2. Define the function $f'_R(Z)$ that computes $f(R^{-1} \cdot Z)$ on the input Z. Use an IFSS scheme to compute $(k_1, \ldots, k_n) \leftarrow \mathtt{Gen}(1^\lambda, f'_R)$.
3. Output k_i to P_i for each $i \in [n]$ and r_α to P_{i_α} for each $\alpha \in [t]$.

The Protocol. The following steps are run in the online phase:
Phase 1 (Round 1 to Round t). *(Input Masking)* In round α, P_{i_α} does the following:

- If $\alpha = 1$, let $V_\alpha = \mathsf{r}_\alpha \cdot \pi_{\alpha,X_{i_\alpha}}$.
- If $\alpha \neq 1$, let $V_{\alpha-1}$ denote the message received from $P_{i_{\alpha-1}}$ during the previous round. Compute $V_\alpha \leftarrow \mathsf{r}_\alpha \cdot V_{\alpha-1} \cdot \pi_{\alpha,X_{i_\alpha}}$.
- If $\alpha < t$, send V_α to $P_{i_{\alpha+1}}$.
- If $\alpha = t$, set $Z = V_t$.

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

- If $i = n$, set $y_i = \mathtt{Eval}(i, k_i, Z, \perp)$.

[15] Note that the group operation need not be multiplication specifically. We use 'product' as a general term to refer to the value obtained by applying the group operation on a sequence of non-abelian group elements.

– If $i \neq n$, parse the message received from P_{i+1} in the previous round as (Z, y_{i+1}). Compute $y_i = \texttt{Eval}(i, k_i, Z, y_{i+1})$.
– If $i \neq 1$ Send (Z, y_i) to P_{i-1}.

Output Computation. P_1 sets the output $y = y_1$.

Phase 3 (Round $t+n+1$ to $t+2n$) *(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 Z computed by P_{i_t} at the end of Phase 1 is $V_t = \mathsf{r}_t \cdot \big(\mathsf{r}_{t-1} \cdot \ldots \cdot (\mathsf{r}_2 \cdot (\mathsf{r}_1 \cdot \pi_{1,X_{i_1}}) \cdot \pi_{2,X_{i_2}}) \cdot \ldots \pi_{t-1,X_{i_{t-1}}}\big) \cdot \pi_{t,X_{i_t}} = (\mathsf{r}_t \cdot \mathsf{r}_{t-1} \cdot \ldots \cdot \mathsf{r}_2 \cdot \mathsf{r}_1) \cdot (\pi_{1,X_{i_1}} \cdot \pi_{2,X_{i_2}} \cdot \ldots \pi_{t-1,X_{i_{t-1}}} \cdot \pi_{t,X_{i_t}}) = R \cdot \pi_{1,X_{i_1}} \cdot \pi_{2,X_{i_2}} \cdot \ldots \pi_{t-1,X_{i_{t-1}}} \cdot \pi_{t,X_{i_t}}$. The IFSS then evaluates $f(R^{-1} \cdot Z) = f(R^{-1} \cdot R \cdot \pi_{1,X_{i_1}} \cdot \pi_{2,X_{i_2}} \cdot \ldots \pi_{t-1,X_{i_{t-1}}} \cdot \pi_{t,X_{i_t}}) = f(\pi_{1,X_{i_1}} \cdot \pi_{2,X_{i_2}} \cdot \ldots \pi_{t-1,X_{i_{t-1}}} \cdot \pi_{t,X_{i_t}})$.

Privacy. Intuitively, security of $\Pi_{\texttt{nonabl}}$ follows from the security of IFSS (that ensures that R remains hidden) and the fact that the values communicated in Phase 1 are indistinguishable from random elements of the group (due to masking with uniformly random elements of the group). The formal security proof of $\Pi_{\texttt{nonabl}}$ is almost identical to the security proof for abelian programs (Thm 1).

$\mathcal{BC}$-Analysis. We analyze the maximum communication complexity incurred by any party in the protocol. In Phase 1, a party may incur communication complexity of at most $\mathcal{O}(k \log |G|)$ (as a party's input may appear at most k times). The $\mathcal{BC}$ analysis of phase 2 and 3 is the same as for abelian programs which totals upto $\mathcal{O}(\lambda \log |G|)$ for GC-based IFSS and $\mathcal{O}(\log |G|)$ for the OTTT-based IFSS. We can thus conclude that the overall $\mathcal{BC}$ of the protocol is $\mathcal{O}(\log |G|(k+\lambda))$ for the computational variant and $\mathcal{O}(k \log |G|)$ for the information-theoretic variant.

The discussion above therefore leads to the following:

Theorem 2. *Protocol $\Pi_{\texttt{nonabl}}$ securely computes the non-abelian program f against a semi-honest adversary corrupting upto $n-1$ parties. The $\mathcal{BC}$ of $\Pi_{\texttt{nonabl}}$ is $\mathcal{O}(k \log |G|)$ and $\mathcal{O}(\log |G|(k+\lambda))$ for the information-theoretic and the computational variant respectively.*

7 $\mathcal{BC}$-efficient MPC for tree-structured circuits

In this section, we build on the ideas of our generalized protocol for abelian programs in Figure 5.1 to formulate $\mathcal{BC}$-efficient protocols for additional classes of functions.

We present a protocol that using IFSS allows to evaluate, in a $\mathcal{BC}$-efficient way, functions that can be expressed as a tree of sub-functions, each taking inputs only from a subset of parties. This can be visualized as a tree with the leaves representing inputs and the non-leaf nodes representing functions. More specifically, the root of a sub-tree would represent the sub-function involving the values in the sub-tree. For our construction to be $\mathcal{BC}$ efficient, we require that each sub-function involves at most $o(n)$ inputs [16]. Further, the resulting tree depth $d = \log_{\mathsf{m}}(n)$ must also be sublinear in n. As a concrete example, the function $g = F\big(f_1(x_1^1, \ldots, x_{\mathsf{m}}^1), \ldots, f_{\mathsf{B}}(x_1^{\mathsf{B}}, \ldots, x_{\mathsf{m}}^{\mathsf{B}})\big)$, can be expressed as 2-level tree with $\mathsf{m} = \sqrt{n}$, where the f_i's denote the sub-functions at level 1 and F denotes the function (represented by the root) that aggregates the outputs of these sub-functions.

The main idea is that the subset of parties involved in computation of a single sub-function first interact among themselves to compute the outputs of this sub-function, which are later aggregated to compute the

[16] However, if there is a $\mathcal{BC}$-efficient protocol independent of the number of inputs (such as our protocol for abelian programs) that can be used to compute the sub-function, then our construction does not require the number of inputs to this sub-function to be sublinear in n.

final output. In order to 'hide' the outputs of these sub-functions (since they may not necessarily be leaked by the output), the sub-functions are tweaked to compute a 'masked' output instead, using a mask chosen by the setup. When multiple 'masked' outputs are taken as inputs to another sub-function (at a higher-level of the tree), the sub-function is further tweaked to first unmasks these values and then compute the function.

We observe that the above approach for an m-ary tree of depth $d = \log_{\mathsf{m}} n$ would result in $\mathcal{BC}$ of $\Omega(\mathsf{m} \log_{\mathsf{m}} n)$ Note that this term $\mathsf{m} \cdot \log_{\mathsf{m}} n$ incurs the least communication when $\mathsf{m} = 2$. Since choosing $\mathsf{m} = 2$ results in better $\mathcal{BC}$-efficiency, we focus only on binary trees in our formal protocol specification.

For ease of exposition, we use a slightly different naming convention and let the n parties be $P_0, \ldots, P_{n-1}$, with $n = 2^d$, and we consider a function $f(x_0, \ldots, x_{n-1})$ that can be decomposed with a binary tree of binary functions as explained below. Note that this can be easily generalized to the k-read setting by letting each party "control" k different inputs of the functions, but doing in the protocol description below would introduce unnecessarily cumbersome notation.

We start by introducing some useful notation to explain our protocol: We label the leaves of the binary tree with the bitstrings corresponding to the indices of the parties i.e., $0^d, 0^{d-1}1, \ldots, 1^d$, and we assign the input of each party to its corresponding leaf. (We will use integers and strings representing them interchangeably in the protocol description i.e., $P_0 = P_{0^d}, P_1 = P_{0^{d-1}1}, \ldots$). The internal nodes of the tree correspond to the functions into which f can be decomposed. To label the internal nodes/functions, we introduce the wildchar $*$, and we label the $n/2$ parents of the leaf nodes as $0^{d-1}*, 0^{d-2}1*, \ldots, 1^{d-1}*$, assigning one function to each such node. We continue introducing an extra wildchar $*$ every time we climb a layer of the tree until we reach the root that gets labeled as $*^d$, corresponding to the function f_{*^d}. For simplicity, we assume that all the inputs and the outputs of all the functions in the tree are elements of the same group G.

We also introduce some notation to deal with strings with wildchars: We say a string $s \in \{0, 1, *\}^d$ is valid if the wildchar $*$ is only followed by other $*$ wildchars (e.g., $0*$ is valid while $*0$ is not). Then, given a valid string s, we denote by $s|b$ the (valid) string s where the first wildchar $*$ is replaced by the bit b. Finally, given a valid string s we define $[s] \subseteq \{0, 1\}^d$ to be the set of all strings that can be obtained when replacing the wildchars $*$ in s with bits.

We can now conveniently describe how to decompose the function $f(x_1, \ldots, x_n)$: for all valid strings $s \in \{0, 1, *\}^d$ (starting with the parents of the leaves) we compute $x_s = f_s(x_{s|0}, x_{s|1})$, and finally we let the output be $f(x_1, \ldots, x_n) = f_{*^d}(x_{*^{d-1}0}, x_{*^{d-1}1}) = x_{*^d}$. In other words, we begin by pairing the leaf inputs two-by-two, then combine the results of these computations two-by-two climbing the tree until we reach the root.

We now need to address two issues in order to evaluate such functions *securely* and in a *$\mathcal{BC}$-efficient* way. First, we need to make sure that no intermediate values are leaked. This can be solved by assigning a mask $\mathsf{r}_{s|b}$ on each edge of the tree, such that the child function $f_{s|b}$ will mask its output with $\mathsf{r}_{s|b}$, and its parent function will de-mask the inputs before evaluating the function. That is, instead of evaluating $f_s(x_{s|0}, x_{s|1})$ we will evaluate using an IFSS scheme

$$f'_s(z_{s|0}, z_{s|1}) = f_s(z_{s|0} - \mathsf{r}_{s|0}, z_{s|1} - \mathsf{r}_{s|1}) + \mathsf{r}_s$$

(where the root has no mask i.e., $\mathsf{r}_{*^d} = 0$). Second, to make sure that the overall protocol is $\mathcal{BC}$-efficient, we will only let the parties P_i with $i \in [s]$ participate in the secure evaluation of f'_s. Intuitively, this is fine since if all parties $i \in [s]$ are corrupt then they would already be able to compute all inputs and outputs in the subtree of the function f_s, thus it does not matter if those masks leak due to the fact that all parties involved in those IFSS computations are corrupt.

We are now ready for the formal description of the protocol. For convenience, we enhance the notation of the IFSS generation algorithm $\mathtt{Gen}$ to include an extra parameter $S \subseteq [n]$, which indicates which subset of parties should receive keys i.e., running $(k_0, \ldots, k_{n-1}) \leftarrow \mathtt{Gen}(1^\lambda, S, f)$ returns $|S|$ IFSS keys k_i for $i \in S$ and sets $k_i = \perp$ for $i \notin S$.

Figure 7.1: $\Pi_{\texttt{tree}}$

Private input. There are $n = 2^d$ parties. Each party P_i, with $i = 0, \ldots, n-1$ has input x_i. We assume all inputs are from some goup G.

Correlated Randomness Setup.

1. For each valid string $s \in \{0, 1, *\}^d$ choose a uniform random mask r_s from G, except for r_{*^d} which is set to 0.
2. For each valid string $s \in \{0, 1, *\}^d \setminus \{0, 1\}^d$ (e.g., for all nodes except the leaves) run the IFSS setup
$$(k_0^s, \ldots, k_{n-1}^s) \leftarrow \texttt{Gen}(1^\lambda, [s], f'_s)$$
(remember that $k_i^s = \perp$ for all $i \notin [s]$) where f'_s is defined as
$$f'_s(z_{s|0}, z_{s|1}) = f_s(z_{s|0} - \mathsf{r}_{s|0}, z_{s|1} - \mathsf{r}_{s|1}) + \mathsf{r}_s$$

Finally, send to each party P_i their mask r_i and the keys k_i^s for all s such that $i \in [s]$.

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

1. *Transferring Masked Inputs for Leaf Nodes.*
Each P_i sets $z_i = x_i + \mathsf{r}_i$ and sends it to their "sibling" party i.e., if $i = s|b$ send z_i to $P_{s|(1-b)}$.
2. *Climbing the Tree.*
For all valid strings $s \in \{0, 1, *\}^d \setminus \{0, 1\}^d$ (e.g., for all intermediate nodes, starting with the parents of the leaves):
   (a) *Evaluating the IFSS.*
   Let all parties P_i with $i \in [s]$ run the IFSS evaluation on inputs $z_{s|0}, z_{s|1}$ e.g., starting from the party with the highest index $i \in [s]$ and going backwards run:
   $$y_i^s = \texttt{Eval}(i, k_i^s, (z_{s|0}, z_{s|1}), y_{i+1}^s)$$
   (where as usual $y_j^s = \perp$ if j is "out of bounds").
   (b) *Transfering Masked Outputs.*
   Let ι be the smallest index in $[s]$. Let all parties P_i with $i \in [s]$ learn the output $z_s = y_\iota^s$. E.g., all parties in $[s]$, starting from P_ι, send z_s to the next party in $[s]$.
   (c) *Transfering Masked Inputs for Subtrees.* Each P_i with $i \in [s]$ sends z_s to one party in the "sibling" sub-tree i.e., if $i = s|b_1, \ldots, b_h$ (with h representing the height we have reached in the tree), then P_i sends z_s to P_j with $j = i = s|(1 - b_1), \ldots, b_h$.

Correctness. Thanks to the correctness of the IFSS scheme the output of each node in the tree is computed correctly, meaning that the input masks are removed by f'_s before evaluating f_s and adding the output mask. Finally, since the mask of the root r_{*^*} is 0, the output of the final computation z_{*^*} is equal to $f(x_0, \ldots, x_{n-1})$.

$\mathcal{BC}$-Analysis. First, we note that to transfer the masked inputs, a party sends messages of size at most $\mathcal{O}(\log |G|)$. Next, consider evaluation of a specific sub-function. Here, transferring masked outputs would require a party to send messages of size at most $\mathcal{O}(\log |G|)$ along a chain. Next, the steps using IFSS incur communication of size at most $\mathcal{O}(\lambda \log |G|)$ for GC-based instantiation and $\mathcal{O}(\log |G|)$ for the OTTT-based instantiation. Since the above occurs for each level and there are $\log(n)$ levels, we can conclude that the overall $\mathcal{BC}$ of the protocol is $\mathcal{O}(\lambda \cdot \log |G| \cdot \log(n))$ for the computational variant and $\mathcal{O}(\log |G| \cdot \log(n))$ for the information-theoretic variant. The above discussion assumes balanced trees. If this is not the case, more generally, for depth d, the $\mathcal{BC}$ is $\mathcal{O}(\lambda \cdot \log |G| \cdot d)$ for the computational variant and $\mathcal{O}(\log |G| \cdot d)$ for the information-theoretic variant.

For a read-k tree-like structure (where a party's input could correspond to at most k leaves), the number of leaves is at most kn and the depth for a balanced tree is $\log(kn)$. This results in $\mathcal{BC}$ of $\mathcal{O}(k \cdot \lambda \cdot \log |G| \cdot \log(kn))$ for the computational variant and $\mathcal{O}(k \cdot \log |G| \cdot \log(kn))$ for the information-theoretic variant.

Privacy. Proving privacy of the tree-based construction requires building a simulator that can simulate the view of an adversary corrupting up to $n-1$ parties in the protocol. This can be done following the blueprint of the simulator of the protocol $\Pi_{\mathtt{abl}}$. That is, the simulator receives as input the output of the computation y as well as the inputs of the malicious parties. The simulator will pick random values for all edges on the tree and simulate the setup phase by running the simulator of the IFSS on those random inputs/outputs. Then, the simulator will provide masks to the adversary which are consistent with these random inputs.

In the online phase, the simulator will simulate the transfer of masked inputs of leaf nodes using the random values already chosen during setup. Then, the simulator includes in the view of the corrupted parties the values y_i^s provided by the IFSS simulator. Indistinguishability between the real protocol and the simulated execution can be argued by replacing, one by one, each real execution of IFSS with a simulated one. Indistinguishability in this first series of hybrids follows from the privacy guarantees of IFSS. In the next series of hybrids, we replace the masked inputs/outputs learned by the adversary in the protocol execution with uniformly random values from G. Since in this hybrid the masks of the honest parties are not used anymore (as the IFSS is simulated), this new series of hybrids are all unconditionally indistinguishable from their previous one. As the final hybrid of this series corresponds to the simulator, this concludes the argument.

The discussion above therefore leads to the following:

Theorem 3. *Protocol $\Pi_{\mathtt{tree}}$ securely computes the aggregated function f against a semi-honest adversary corrupting upto $n-1$. The $\mathcal{BC}$ of $\Pi_{\mathtt{tree}}$ is $\mathcal{O}(k \cdot \log|G| \cdot \log(kn))$ and $\mathcal{O}(k \cdot \lambda \cdot \log|G| \cdot \log(kn))$ for the information-theoretic and computational variant respectively.*

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.

Bar85. David Arno Barrington. *Width-3 permutation branching programs.* Laboratory for Computer Science, Massachusetts Institute of Technology, 1985.

Bar86. David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc(1). In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 1–5. ACM, 1986.

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.

BGI16. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1292–1303. ACM Press, October 2016.

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.

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.

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.

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.

FGKS22. Rex Fernando, Yuval Gelles, Ilan Komargodski, and Elaine Shi. Maliciously secure massively parallel computation for all-but-one corruptions. In *CRYPTO 2022*, 2022.

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, Part II*, volume 12551 of *LNCS*, pages 379–409. Springer, Heidelberg, November 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.

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

ORS22. Claudio Orlandi, Divya Ravi, and Peter Scholl. On the bottleneck complexity of mpc with correlated randomness. *International Conference on Practice and Theory of Public-Key Cryptography*, 2022.

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.

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.