Secure Multiparty Computation with Free Branching

Aarushi Goel[1], Mathias Hall-Andersen[2], Aditya Hegde[1], and Abhishek Jain[1]

[1] Johns Hopkins University, Baltimore, USA,
{aarushig,ahegde,abhishek}@cs.jhu.edu,
[2] Aarhus University, Aarhus, Denmark, ma@cs.au.dk

Abstract. We study secure multi-party computation (MPC) protocols for branching circuits that contain multiple sub-circuits (i.e., branches) and the output of the circuit is that of single "active" branch. Crucially, the identity of the active branch must remain hidden from the protocol participants.

While such circuits can be securely computed by evaluating each branch and then multiplexing the output, such an approach incurs a communication cost linear in the size of the entire circuit. To alleviate this, a series of recent works have investigated the problem of reducing the communication cost of branching executions inside MPC (without relying on fully homomorphic encryption). Most notably, the stacked garbling paradigm [Heath and Kolesnikov, CRYPTO'20] yields garbled circuits for branching circuits whose size only depends on the size of the largest branch. Presently, however, it is not known how to obtain similar communication improvements for secure computation involving *more than two parties*.

In this work, we provide a generic framework for branching multi-party computation that supports *any number of parties*. The communication complexity of our scheme is proportional to the size of the largest branch and the computation is linear in the size of the entire circuit. We provide an implementation and benchmarks to demonstrate practicality of our approach.

1 Introduction

Secure multiparty computation (MPC) [40,20,9,5] is an interactive protocol that allows a group of mutually distrusting parties to jointly compute a function over their private inputs without revealing anything beyond the output of the function. Over the years, significant progress has been made towards improving the efficiency of MPC protocols [11,39,24,21,3,23,33,25,12,22,16] to make them practically viable.

While a wide variety of techniques for efficiency improvements have been developed in different settings based on the corruption threshold, communication model or security guarantee, a common aspect of most modern efficient protocols in all of these settings is that they *circuit representation* of the function. A limitation of such protocols, however, is that their total communication complexity

is at least linear in the size of the circuit. Known techniques for getting sub-linear communication in the circuit size rely on computationally heavy tools such as fully-homomorphic encryption (FHE) [19] or homomorphic secret sharing (HSS) [7]. While there have been recent advancements in improving the efficiency of these methods, they are still far from being practical in many use cases.

As a result, the efficiency of existing efficient protocols is highly dependent on how succinctly a function can be represented using circuits. This is clearly not ideal, since circuits are often not the most efficient way of representing many functions. A common example of such functions are ones that include some kind of *conditional* control flow instructions. When evaluating such functions, a circuit-based MPC will incur communication dependent on the size of the *entire* circuit, while in reality we only need to evaluate the "active" path (i.e., the path that is actually executed based on the conditional) in the circuit.

It is therefore useful to design efficient MPC protocols for useful classes of functions, where the total communication between the parties only depends on the "active" parts, rather than the entire circuit.

MPC for Conditional Branches. In this work, we focus on one such class of functions, namely, ones that contain conditional branches. As discussed in [29], a real world example of an application that consists of conditional branches is where a set of servers collectively provide k services and the clients can pay and avail any one of their services (depending on their requirements), without revealing to the servers which service they are availing. Similarly, control flow instructions are also integral to any kind of programming and as observed in [27], many kinds of control flow instructions (including repeated and/or nested loops) can be refactored into conditional branches. Such refactorings often result in a *large* number of conditional branches. For such functions, designing MPC protocols where the total communication only depends on the size of the active branch is very useful.

Recently, in a sequence of works [26,27], Heath and Kolsnekov made progress in this direction in the two-party setting. They design garbled circuit based *two-party* semi-honest protocols for evaluating functions with conditional branches, where the total communication only depends on the size of the largest branch. In the *multiparty* setting, however, no such protocols are currently known. The recent works of [28,30] design MPC for conditional branches where they reduce the number of public-key operations required to evaluate conditional branches; however, the total communication in their protocols still depends on the size of all branches. Furthermore, all these protocols only work for Boolean circuits.

Given this state of the art, we consider the following question in this work:

Does there exist an efficient multiparty protocol for securely computing conditional branches, where the total communication only depends on the size of the largest branch?

We remark that all of the above mentioned prior works only focus on the semi-honest setting. The task of designing analogous *maliciously-secure* protocols

remains unexplored (both in the two-party and multi-party settings). In this work, we also consider this question.

1.1 Our Contributions

We design the first *multiparty* computation protocols for conditional branches, where the communication complexity only depends on the size of the largest branch. Our protocols can support arbitrary number of parties and corruptions. We present both constant and non-constant round variants.

I. Non-Constant Round Branching MPC. Our first contribution is a *semi-honest* MPC for conditional branches, where the communication complexity only depends on the size of the largest branch. This protocol is capable of computing arithmetic circuits over any field or ring. The round complexity of this protocol depends on the depth of the circuit.

We present this protocol as a generic compiler that can transform a large class of admissible[3] MPC protocols into ones for conditional branches that achieve the aforementioned communication complexity. Several existing concretely efficient protocols including MASCOT [33], SPDZ2k [12], Overdrive [34], TinyOT [18] and [25], [13] can be used with this compiler.

In particular, by instantiating our compiler with a semi-honest admissible (dishonest-majority) MPC protocol with communication complexity $\mathtt{CC}(|C|)$ (where C is the circuit being evaluated), we obtain the following result:

Informal Theorem 1 *Let λ be the security parameter. There exists a semi-honest secure MPC for evaluating conditional branches, that can tolerate arbitrary corruptions and that achieves communication complexity of $O(\mathtt{CC}(|C_{\mathsf{max}}|) + n^2 k\lambda + n^2|C_{\mathsf{max}}|)$, where k is the number of branches in the conditional.*

We also implement this protocol to test its concrete efficiency and compare it to state-of-the-art MPC protocols. More details are provided later in this section.

Extension to Malicious Security. We also present an extension of this protocol to the case of malicious adversaries. Asymptotically, its communication complexity is similar to the semi-honest protocol, except that it incurs a multiplicative overhead dependent on a statistical security parameter.

We view this construction as initial evidence that efficient branching MPC with malicious security is possible. However, we believe that there is significant scope for future improvements towards achieving good concrete efficiency.

II. Constant Round Branching MPC. Our next contribution is a *constant round* MPC for conditional branches, where the communication complexity only depends on the size of the largest branch. This protocol is based on a *multiparty garbling* approach [2] and only supports boolean circuits.

[3] We require the underlying MPC to be such that it evaluates the circuit in a gate-by-gate manner and maintains an invariant that for every intermediate wire in the circuit, the parties collectively hold a sharing of the value induced on that wire during evaluation.

We also present this protocol in the form of a general compiler. Namely, given a MPC protocol with communication complexity $\mathtt{CC}(|C|)$ for evaluating a circuit C, we get the following result:

Informal Theorem 2 *Let λ be the security parameter. There exists a constant-round, semi-honest secure MPC for evaluating conditional branches (represented as Boolean circuits), that can tolerate arbitrary corruptions and that achieves communication complexity of $O(|\mathtt{CC}(\lambda|C_{\mathsf{max}}|) + n^2k\lambda + n^2\lambda|C_{\mathsf{max}}|)$, where k is the number of branches in the conditional.*

To obtain both of the above results, we adopt a fundamentally different approach as compared to prior works [26,28,27,30] in this area. Specifically, prior works require the parties to locally evaluate *all* the branches. In contrast, in our approach, the parties select the "active" branch and *only execute that branch*. A detailed overview our approach can be found in the next section.

III. Comparison and Performance Evaluation. To gauge practicality, we implement our non-constant round *semi-honest* compiler and instantiate it using two kinds of protocols:

- *Quadratic Dependence on the Number of Parties:* MP-SPDZ is a common MPC library that contains implementations of the SPDZ protocol [16] and its descendants. All of the protocols in this library have total communication with quadratic dependence on the number of parties. We instatiate our compiler with an implementation of MASCOT [33] from this library without modification. Our code is agnostic to which protocol the MPC library is configured; this helps demonstrates that our techniques are generic and block-box. We run benchmarks over simulated LAN and WAN settings. We show that our compiled protocol outperforms naïvely evaluating all the branches in parallel using MASCOT for as few as 8 branches.
- *Linear Dependence on the Number of Parties:* We implement an optimized variant of our compiler that incurs a linear additive overhead in the number of parties, instead of a quadratic overhead. We then test the efficiency of our compiler when instantiated with the CDN protocol [13], which only has a linear dependence on the number of parties. For this, we first implement the CDN protocol. To the best of our knowledge, this is the first known implementation of CDN. Similar to the previous case, we show that our compiled protocol (instantiated using CDN) outperforms naïvely evaluating all the branches in parallel using CDN for 8 branches.

2 Technical Overview

Background. All recent works [26,28,27,30] in this area are based on the same principle approach – *the parties evaluate all branches, albeit, only the "active" branch is evaluated on real inputs, while the remaining branches are all evaluated on fake/garbage values.*

For instance, in the two-party setting, [26,27], which are based on a garbled circuit based approach, one of the parties garbles all the k branches. It then "stacks" these garblings into a compressed form that is proportional to the length of the largest branch in the circuit. Using some additional information sent by the garbler, the evaluator is able to reconstruct k different garbled circuits, only one of which is a valid garbling of the "active" branch, and the remaining are random strings (or some garbage material). Unaware of the active branch, the evaluator evaluates the k garbled circuits w.r.t. different branches to obtain k different output labels. These output labels are then filtered with the help of a "multiplexer" to obtain the correct output. Overall, this approach reduces the communication to only depend on the size of the largest branch (the computation complexity, however, is still large).

In the multiparty setting, both [28,30], follow the same principle approach. These protocols have separate preprocessing and online phases. They require parties to evaluate all branches (including the inactive ones) in the online phase over 0 or some random values and leverage this fact to get savings in the preprocessing phase. As a result, communication in the preprocessing phase only depends on the size of one branch, but the communication in the online phase still depends on the size of all the branches.

Indeed, it is unclear how to extend the stacked garbling approach used in [26,27] to get similar savings in communication in the multiparty setting. Recall that the garbler in stacked garbling is required to garble all branches and hence its computation *depends on the size of all branches*. This means that naive approaches that involve distributing the role of the garbler amongst multiple parties are a non-starter as they will incur *communication* proportional to the size of all branches. In order to design a multiparty protocol with similar communication savings as in stacked garbling, we therefore adopt a fundamentally different approach.

Our Approach. In our approach, *the parties select which branch to execute in a "privacy-preserving" manner and only execute that branch.* To facilitate this private selection, both of our constructions (in the non-constant round and constant-round settings) employ a common tool – a variant of oblivious linear evaluation that we refer to as *oblivious inner product* (OIP). In particular, our protocols make use of OIPs with (small) *constant rate*. We show that such OIPs can be easily constructed using low-rate linearly homomorphic encryption schemes, which are known from a variety of assumptions [17,8,15,38].

In the sequel, we first describe the main ideas underlying our non-constant round constructions. We then proceed to describe our constant-round construction.

2.1 Non-Constant Round Branching MPC

We start with the observation that the problem of computing conditional branches bears some similarities to the problem of private function evaluation (PFE) [31,35,36]. Recall that in PFE, one party has the function and the remaining parties provide inputs. This, in some sense is reminiscent of the problem that

we have at hand, albeit with some differences. In particular, in our case, while none of the parties actually knows which function/branch is "active", they all know the set that this branch belongs to. Moreover, the parties collectively hold information about which of these functions to evaluate. This can be viewed as a *distributed variant of PFE*. In light of this observation, we build upon some ideas previously used in the PFE literature.

Private Function Evaluation. In PFE, the function is only known to one of the parties (say party P_1). The security requirements in standard PFE are very similar to that in MPC, with the only additional requirement that the function must remain hidden from all other parties. To achieve this, Mohassel and Sadeghian [35] observe that in order to hide a function that is represented in the form of a circuit, there are two components that need to remain hidden – (1) The wire-configuration of the circuit, i.e., how the gates connect with each other, and (2) the function (i.e., addition or multiplication) implemented by each gate in the circuit. They propose a strategy to conceal the above components of a circuit in order to achieve function privacy (without relying on universal circuits). In particular, they start with MPC protocols that work over some kind of secret shares (additive/threshold/authenticated) and evaluate any given circuit in a gate-by-gate manner. These protocols maintain the invariant that for every intermediate wire in the circuit, all parties hold a sharing of the value induced on that wire during evaluation. Many concretely efficient protocols such as [33,25,12,22,16], satisfy these requirements. [35] propose the following modifications to such MPC protocols to obtain a PFE protocol:

1. *Hiding Wire Configuration:* Each intermediate wire in the circuit has two end points – (1) one is the *source gate*, for which it acts as the outgoing wire and (2) the other is the *destination gate*, for which it acts as the incoming wire. As discussed earlier, for hiding the wire configuration, we need to hide the gate connections, i.e., we want to hide the mapping between the source and destination of each wire in the circuit. For this, [35] assign two unique labels to each wire w. One is an outgoing label based on its source gate and second is an incoming label based on whether it acts as left or right input wire to its destination gate. Let π denote the mapping between these incoming and outgoing labels, i.e., let $\pi(i) = j$ denote that a wire that has incoming label i has an outgoing label j. In PFE, this mapping π is only known to the function folding party.
   In order to hide this mapping, [35] devise a mechanism to mask the outputs value of each gate and unmask them based on π when this value is used for evaluating the destination gate of this wire. This is executed by sampling an input mask and an output mask for every wire in the circuit. Let $\mathsf{in}_1, \ldots, \mathsf{in}_W$ and $\mathsf{out}_1, \ldots, \mathsf{out}_W$ be the set of these input and output masks, where W is the total number of wires in the circuit. In the preprocessing phase, with the help of the function holding party and the underlyin MPC, the parties compute $\Delta_w = \mathsf{in}_w - \mathsf{out}_{\pi(w)}$ for every $w \in [W]$. These Δ_w values are revealed to function holding party in the clear. This processing information helps the parties in using appropriately permuted input and output masks

to mask and unmask wire values during evaluation in the online phase. In more detail, the online phase proceeds as follows:

- Upon evaluating each gate g, the parties use output masks to mask all the outgoing wires of the gate. Let the outgoing wires have labels c and d respectively, and let u_c and u_d denote these masked outputs. These masked outputs are revealed to all parties in the clear.
- For evaluating a particular gate g, where the two input wires have incoming wire labels a and b, the function holding party computes $A = u_{\pi(a)} + \Delta_a$ and $B = u_{\pi(b)} + \Delta_b$ and sends it to all the parties. The parties subtract their shares of inp_a and inp_b from these values to get a sharing of the actual values on which to evaluate gate g.

2. *Hiding Gate Functions:* This is relatively easier. Assume that our arithmetic circuit representation of the function only consists of addition and multiplication gates, let $\mathsf{type}_g = 0$ (and $\mathsf{type}_g = 1$ resp.) denote that gate g is an addition gate (and multiplication gate resp.). For each gate g with incoming wires a and b, we can use the underlying MPC to compute both shares of $a + b$ and $a \cdot b$. The function holding party P_1 can secret share type_g using the underlying MPC and the parties can then choose between shares of $a+b$ and $a \cdot b$ by computing the following using the underlying MPC:

$$(1 - [\mathsf{type}_g])([a + b]) + \mathsf{type}_g([a \cdot b]),$$

where we denote $[x]$ as a sharing of a value x using the secret sharing scheme used by the underlying MPC. This allows the parties to evaluate the correct function, without revealing it.

Our Semi-Honest Protocol. In our setting, the parties know the description of all the branches in the conditional and have a secret sharing of the index of the active branch. In order to hide the identity of the active branch, similar to the above approach, we need to hide both the wire configuration and the gate functions of the active branch. We start by listing the barriers in directly adapting the above approach to our setting and then proceed to discuss how we resolve them.

- In the preprocessing phase, computing Δ requires the function holding party to input π to the underlying MPC. In our setting, no party knows the exact value of π.
- In the online phase, A and B values are computed locally by the function holding party in PFE since it already knows the mapping π. This is again a problem in our setting.
- Finally, in order to hide the gate functions in the online phase, the value of each type_g secret shared by the function holding party. But as above, neither party in our setting knows this value.

In order to overcome the above barriers, we crucially rely on the fact that in our setting, while no single party knows the function (or the mapping π), they all know the set that the function belongs to. In other words, given a set of k

branches $C_1, \ldots, C_k$, all the parties can locally compute the mappings $\pi_1, \ldots, \pi_k$ corresponding to each branch. Moreover, the parties also have a secret sharing of the index of the active branch. Let α be the index of the active branch. Our first idea towards resolving the above barriers to is to somehow allow the parties combine their shares of α with $\pi_1, \ldots, \pi_k$ to get a sharing of π_α. However, since the size of $\pi_1, \ldots, \pi_k$ depends on the size of all branches, a naive implementation of this computation will incur communication that depends on the size of $\pi_1, \ldots, \pi_k$.

We get around this by using a new variant of oblivious linear evaluation, which we refer to as oblivious inner product. We now outline our main ideas:

- **Sharing of α:** We work with a unary representation of the index α. In other words, we assume each party have k secret shares, where the α^{th} share is a sharing of 1, while all others are sharings of 0s. Let these shares be denoted by $[b_1], \ldots, [b_k]$
- **Input/Output Masks:** In the preprocessing phase, we use the underlying MPC to sample random input and output masks $\mathsf{in}_1, \ldots, \mathsf{in}_W$ and $\mathsf{out}_1, \ldots, \mathsf{out}_W$, where W is the number of wires in the largest branch. Each party, now locally permutes its shares of input masks based on the k mappings $\pi_1, \ldots, \pi_k$. In more detail, given sharings $[\mathsf{out}_1], \ldots, [\mathsf{out}_W]$, for each $m \in [k]$, the parties locally compute sharings $[\mathsf{out}_{\pi_m(1)}], \ldots, [\mathsf{out}_{\pi_m(W)}]$. Lets denote each $[\mathsf{out}_{\pi_m(1)}], \ldots, [\mathsf{out}_{\pi_m(W)}]$ by $[\overrightarrow{\mathsf{out}_{\pi_m}}]$. If instead of computing shares of π_α, we directly compute re-randomized shares of $[\overrightarrow{\mathsf{out}_{\pi_\alpha}}]$, then the parties can simply compute their shares of Δ_w values as follows

$$\forall w \in [W], [\Delta_w] = [\mathsf{in}_w] - [\mathsf{out}_{\pi_\alpha(w)}]$$

- **Oblivious Inner Product:** For computing re-randomized shares $[\overrightarrow{\mathsf{out}_{\pi_\alpha}}]$, we use a primitive called oblivious inner product (OIP). This is a protocol between two-parties, called the sender and receiver and bears resemblance to oblivious linear evaluation. The sender has inputs $m_0, \ldots, m_k$ and the receiver has inputs $b_1, \ldots, b_k$. At the end of the protocol, the receiver learns $m_0 + \sum_{i \in [k]} b_i m_i$ and the sender learns nothing.
  We use this primitive and a GMW [20] style approach to obtain shares of $\overrightarrow{\mathsf{out}_{\pi_\alpha}}$ as follows: for each pair of parties in the protocol, we run an instance of OIP, where one party acts as the sender and the other acts as the receiver. The inputs of the sender party to this OIP are its shares of $[\overrightarrow{\mathsf{out}_{\pi_1}}], \ldots, [\overrightarrow{\mathsf{out}_{\pi_W}}]$ and a random value X, while the inputs of the receiver are its shares of the unary representation of α. At the end, each party P_i computes its share of $\overrightarrow{\mathsf{out}_{\pi_\alpha}}$ by adding the outputs of each OIP instance where it acted as the receiver and subtracting each X sampled in the OIP instance where it acted as the sender. It is easy to see that these resulting shares are indeed shares of $\overrightarrow{\mathsf{out}_{\pi_\alpha}}$.
  However, note that while the length of the output of each OIP in our case only depends on the size of the largest branch, the length of sender inputs depends on the size of all branches. Therefore, in order to design an MPC

protocol where the overall communication is only proportional to the size of the largest branch, we must use an OIP where the communication only depends on the length of receiver inputs and the output, but is independent of the length of sender inputs. We show that such OIPs can be constructed using linearly homomorphic encryption with constant rate.

– **Online Phase:** Now that we have sharing of Δ_w values that was computed using the mapping π corresponding to the active branch, we can compute shares of the A and B values as follows:

$$[A] = \sum_{m \in [k]} [b_1] u_{\pi_1(a)} + [\Delta_a] \quad \text{and} \quad [B] = \sum_{m \in [k]} [b_1] u_{\pi_1(b)} + [\Delta_b]$$

We note that most linearly homomorphic secret sharing schemes allow such computations to be done non-interactively and hence it does not incur any overhead in the communication complexity. Shares of type_g for every gate g can also be computed in a similar manner.

We present a formal description of this protocol in Section 5.

Extension to Malicious Security. While the basic outline of our protocol remains the same, even in the malicious setting, we need to do a little more work to make the above protocol secure against a malicious adversary. In particular, we need to ensure that the inputs used by the parties in the OIP instances are consistent with values/shares computed by them using the underlying MPC. For this we propose to add the following consistency checks:

Receiver's Input Consistency. We start by using an OIP that is secure against a malicious reciever. In order to ensure that receiver uses valid sharings of the active branch, we implement a kind of MAC check using the underlying MPC. In particular, in the OIP execution, the sender samples $k+1$ random values and appends them to its inputs. Now when the receiver computes the output of the OIP, it also learns an inner product of these random values with its shares of the active branch (we refer to this as the MAC value for this OIP). We now use the underlying MPC to compute the exact same value. In particular, the sender sends the $k+1$ random values that it sampled in the OIP as input to the underlying MPC, while the receiver sends the MAC value learnt from the output of the OIP. We allow the underlying MPC to now check if the MAC value indeed corresponds to an inner product of the receivers shares of the active branch and the random values input by the sender. We note that since the length of the receiver's input is independent of the size of all branches, computing this MAC value inside the MPC does not incur too much overhead.

Sender's Input Consistency. Recall that the inputs of the sender to the OIP depend on the size of all branches, and hence we cannot hope to use the kind of check that we used for ensuring receiver consistency. Moreover, since the length of the sender message is much shorter than the length of its inputs, we also cannot hope to use an OIP with malicious sender security that can somehow extract the sender's inputs. Therefore, instead we continue to work with an

OIP that is secure against a semi-honest sender but augment it with a cut-and-choose style approach. In particular, we sample multiple copies of the masks and compute delta values using OIPs for each of those copies. We also ask the sender to commit (using compressive commitments) to the inputs and randomness used for computing each of its sender messages. At the end of all OIP instantiations, we use the underlying MPC to sample a random subset and reveal the shares of masks of all parties for that subset. The senders also send the randomness used by them in the sender messages of this opened subset. Given this information, the parties can verify if the senders behaved honestly and used consistent shares in the opened instances. We use the remaining unopened instances to run multiple copies of the online phase and take a majority to decide the final output. Due to the use of cut-and-choose, the communication complexity of our maliciously secure protocol is proportional to $\delta \times$ the cost of computing the largest branch. We defer a formal description of this protocol to the full-version of this paper.

2.2 Constant Round (Semi-Honest) Protocol

Beaver, Micali, and Rogaway (BMR) [2] proposed a general template for constructing *constant round* MPC from existing generic *non-constant round* MPC. The main observations underlying their technique were – (1) round complexity of more generic non-constant round protocols depends on the depth of the function being computed and (2) garbling [40] a functionality/circuit is a constant depth procedure.

The parties can leverage these observations to first execute a *garbling phase*, where they compute a garbled circuit of the function (that they wished to evaluate) using the non-constant round protocol. This phase will require a constant number of rounds. Given this garbled circuit, they then proceed to the *evaluation phase*, where each party locally evaluates the garbled circuit to learn the output. This phase requires no interaction and hence the overall protocol runs in a constant number of rounds.

More concretely, in the garbling phase, the parties collectively sample two keys $\mathsf{k}_{w,0}, \mathsf{k}_{w,1}$ for every wire w in the circuit. The garbled table for each gate g in the circuit with incoming wires a, b and outgoing wire c, consists of the following four rows, corresponding to $\alpha, \beta \in \{0, 1\}$:

$$ct_{\alpha,\beta} = \mathsf{PRF}_{\mathsf{k}_{a,\alpha}}(g) + \mathsf{PRF}_{\mathsf{k}_{b,\beta}}(g) + \mathsf{k}_{c,g(\alpha,\beta)}$$

Branching MPC using BMR Template. The generality of the BMR approach immediately makes it compatible with our non-constant round semi-honest protocol (from Section 2.1). Indeed, in the garbling phase, parties can use that protocol to compute a garbled circuit for the active branch. During the evaluation phase, however, since the parties do not know which branch the garbled circuit corresponds to, they can evaluate it for every branch and obtain the corresponding output wire labels. Note that only the labels obtained by evaluating w.r.t. to the active branch actually correspond to a *valid set of abels*. Finally, via interaction, parties can determine the output corresponding to the

"valid" set of output labels. The complexity of this last step is independent of the circuit size and only depends on the number of branches times the output length.

While this yields a simple baseline constant round MPC for conditional branches, it is highly inefficient. Since no party knows the keys $\mathsf{k}_{a,\alpha}, \mathsf{k}_{b,\beta}$ in their entirety, they must evaluate the PRF (on these keys) inside an MPC protocol. Since, the circuit representations of PRF's are typically massive, this protocol is unlikely to be concretely efficient. As such, for concrete efficiency, we require a protocol that only makes a *black-box* use of cryptography.

Towards Black-Box use of Cryptography. Damgård and Ishai [14] proposed a variant of the above BMR template that enables parties to evaluate the PRF outside the MPC, thereby only making a black-box use of cryptography.

Specifically, in their approach, each party P_i samples two keys $\mathsf{k}^i_{w,0}, \mathsf{k}^i_{w,1}$ for every wire w in the circuit. In other words, the cumulative keys associated with every wire is a concatenation of all the parties' keys. The garbled table for each gate g in the circuit with incoming wires a, b and outgoing wire c, consists of the following $4 \cdot n$ rows, corresponding to $\alpha, \beta \in \{0,1\}$ and $i \in [n]$:

$$ct^i_{\alpha,\beta} = \bigoplus_{m=1}^{n} \mathsf{PRF}_{\mathsf{k}^m_{a,\alpha}}(g\|i) \;+\; \bigoplus_{m=1}^{n} \mathsf{PRF}_{\mathsf{k}^m_{b,\beta}}(g\|i) + \mathsf{k}^i_{c,g(\alpha,\beta)}$$

It is easy to see that unlike the BMR approach, here the parties are only required to evaluate the PRF on their own keys, which can be done locally and the resulting PRF evaluation can be fed as input to the underlying MPC implementing the garbling functionality.

In our setting, however, this approach posits a fundamental barrier. Recall that for evaluating conditional branches, we want to garble the active branch without revealing the index of the active branch. For this, while garbling any gate (say the j^{th} gate), it is imperative that the parties remain oblivious to both the functionality associated with it and its incoming and outgoing wires. As a result, the parties are unaware of which keys $\mathsf{k}^i_{a,\alpha}, \mathsf{k}^i_{b,\beta}$ to use for computing the corresponding ciphertexts, and hence cannot evaluate the PRF on those keys *locally*. A natural approach to overcome this problem is to perform this evaluation within an MPC; however, we are then back to the realm of non-black-box use of cryptography. As such it is unclear how to directly adapt this approach to our setting, while making a black-box use of cryptography.

Garbling using Key-Homomorphic PRFs. To overcome the above barrier, we explore the work of Ben-Efraim et al. [4] who presented an alternative template for multiparty garbling, using *key-homomorphic* PRFs. These are PRFs with the following property: $\mathsf{PRF}_{\mathsf{k}_1}(m) \tilde{+} \mathsf{PRF}_{\mathsf{k}_2}(m) = \mathsf{PRF}_{\mathsf{k}_1 \tilde{-} \mathsf{k}_2}(m)$, where $\tilde{+}$ and $\tilde{-}$ are some operations. As before, each party samples two keys for every wire in the circuit and given such a PRF, the parties the compute each ciphertext as follows:

$$ct_{\alpha,\beta} = \widetilde{\sum}_{m\in[n]} \left(\mathsf{PRF}_{\mathsf{k}^m_{a,\alpha}}(g) \tilde{+} \mathsf{PRF}_{\mathsf{k}^m_{b,\beta}}(g)\right) \tilde{+} \left(\widetilde{\prod}_{m\in[n]} \mathsf{k}^m_{c,g(\alpha,\beta)}\right)$$

It is easy to see that similar to the previous approach, each party here is only required to evaluate the PRF on its own key, which can be done locally. At first, it might seem that in our setting, the same problem (as before) still persists. Indeed, for local PRF evaluation, the parties are required to know which key to use, which as discussed earlier is not possible when the parties are required to obliviously garble one of the conditional branches. However, we observe that homomorphism of the PRF can be leveraged here to resolve this problem.

Lets assume that the parties start by ordering the gates and wires in every branch in some canonical order. Now, when garbling the j^{th} gate of the active branch, they must choose the appropriate keys from all the keys associated with the j^{th} gate in every branch. We also assume that the parties have a sharing of the unary representation of the index associated with the active branch. The parties can now use multiple instances of OIP (as in our non-constant round protocols) to obtain shares of the keys associated with the two incoming wires of the j^{th} gate in the active branch.

Consider a key homomorphic PRF where both $\tilde{+}$ and $\tilde{\cdot}$ are the same operation associated with the reconstruction algorithm of the secret sharing scheme used in the undelying MPC, i.e., $[\mathsf{PRF}_{\mathsf{k}}(m)] = \mathsf{PRF}_{[\mathsf{k}]}(m)$. This PRF can now be used along with the above observation to compute a garbling of the active branch as follows: for simplicity let's assume that each branch is of the same size and has W wires. The parties start by collectively sampling $2W$ keys. For garbling the j^{th} gate, for each $\alpha, \beta \in \{0,1\}$, they use OIPs to compute shares $[\mathsf{k}_{a,\alpha}], [\mathsf{k}_{b,\beta}]$ and $[\mathsf{k}_{c,g(\alpha,\beta)}]$, where a, b are the incoming and c is the outgoing wire of the j^{th} gate in the active branch and g is the function computed by this gate. Parties can now locally evaluate the PRF on these shares and use the underlying MPC to compute shares of the ciphertexts as follows:

$$[ct_{\alpha,\beta}] = \mathsf{PRF}_{[\mathsf{k}_{a,\alpha}]}(j) + \mathsf{PRF}_{[\mathsf{k}_{b,\beta}]}(j) + [\mathsf{k}_{c,g(\alpha,\beta)}]$$

Upon computing this garbled circuit for the active branch, similar to the baseline solution, parties evaluate it w.r.t. all the branches and then run a "mini-MPC" to filter out the valid labels and determine the final output.

Instantiating Key Homomorphic PRF. Most existing dishonest majority MPC protocols [33,25,12,22,16] use additive secret sharing. To use the above ideas with such protocols, we need an additively key-homomorphic PRF, i.e., where $\mathsf{PRF}_{\mathsf{k}_1}(m) + \mathsf{PRF}_{\mathsf{k}_2}(m) = \mathsf{PRF}_{\mathsf{k}_1+\mathsf{k}_2}(m)$. Unfortunately, key homomorphic PRFs are currently only known from the DDH assumption [37,6] and those PRFs do not achieve a similar additive homomorphism.

Ben-Efraim et al. [4] observed that instead of a PRF, it suffices to use a (decisional) ring LWE based random function here. This function is of the form: $F = f_k : \mathcal{R}_p \to \mathcal{R}_p | f_k(a) = a \cdot k + e$, where $p = 2N + 1$ is a prime, N is a power

of two, $\mathcal{R}_p = \mathbb{Z}_p[X]/(X^N + 1)$ and a, k, and e are polynomials in the ring and the coefficients of e come from a gaussian distribuition. Since a is public, it is easy to see that given additive shares of the key k and error e, it is possible for the parties to locally compute shares of the above function. As is standard when using LWE/RLWE, encrypting using such a random function typically requires multiplying the message (before adding it to the output of this function) with the size of the range from which the message comes from. In the case of garbling, since both the message and keys come from the same distribution, as shown in [4], this requires choosing the parameters carefully and additionally requires sampling the keys from a gaussian distribution. However, since the parties only need to compute additive shares of these keys and errors, this can be done easily by requiring the parties to sample their shares from appropriate distributions. We defer more details to Section 6.

3 Oblivious Inner Product

In this section, we define a variant of oblivious linear evaluation (OLE), which we refer to as *oblivious inner product* (OIP). OIP is a protocol between two parties, called the *sender* and *receiver* respectively. The sender has inputs $(\overrightarrow{m_0}, \ldots, \overrightarrow{m_k})$ in some domain (say $\mathcal{D}^m$), and receiver has inputs $(b_1, \ldots, b_k)$ in the same domain $\mathcal{D}$. At the end of the protocol, the receiver should learn $\overrightarrow{m_0} + \sum_{i\in[k]} b_i \overrightarrow{m_i}$ and nothing more, while the sender should learn nothing about the reciever inputs $b_1, \ldots, b_k$.

For our constructions, we consider two variants of OIP, a *semi-honest* version and one that is secure against a *malicious receiver*. We now define the syntax and the security guarantees of a two-message OIP protocol in the plain model. The definitions can be naturally extended to the CRS model.

Definition 1 (Two-Message Oblivious Inner Product). *A two-message oblivious inner product between a receiver* R *and a sender* S *is defined by a tuple of 3 PPT algorithms* $(\mathsf{OIP}_\mathsf{R}, \mathsf{OIP}_\mathsf{S}, \mathsf{OIP}_\mathsf{out})$*. Let* λ *be the security parameter. The receiver computes* $\mathsf{msg}_\mathsf{R}, \rho$ *as the evaluation of* $\mathsf{OIP}_\mathsf{R}(1^\lambda, (b_1, \ldots, b_k))$*, where* $(b_1, \ldots, b_k) \in \mathcal{D}^k$ *is the receiver's input. The receiver sends* msg_R *to the sender. The sender then computes* msg_S *as the evaluation of* $\mathsf{OIP}_\mathsf{S}(1^\lambda, \mathsf{msg}_\mathsf{R}, (\overrightarrow{m_0}, \ldots, \overrightarrow{m_k}))$*, where* $(\overrightarrow{m_0}, \ldots, \overrightarrow{m_k}) \in \mathcal{D}^{m\times(k+1)}$ *are sender's inputs. The sender sends* msg_S *to the receiver. Finally, the receiver computes the output by evaluating* $\mathsf{OIP}_\mathsf{out}(\rho, \mathsf{msg}_\mathsf{R}, \mathsf{msg}_\mathsf{S})$.

A semi-honest OIP satisfies correctness, security against semi-honest receiver and semi-honest sender*, while the malicious variant satisfies* correctness, security against semi-honest sender and malicious receiver*, which are defined as follows:*

- ***Correctness****: For each* $(\overrightarrow{m_0}, \ldots, \overrightarrow{m_k}) \in \mathcal{D}^{m\times(k+1)}$ *and* $(b_1, \ldots, b_k) \in \mathcal{D}^k$*, the following holds*

$$\Pr\left[\begin{array}{r}(\rho, \mathsf{msg}_\mathsf{R}) \leftarrow \mathsf{OIP}_\mathsf{R}\left(1^\lambda, (b_1, \ldots, b_k)\right) \\ \mathsf{msg}_\mathsf{S} \leftarrow \mathsf{OIP}_\mathsf{S}\left(1^\lambda, \mathsf{msg}_\mathsf{R}, (\overrightarrow{m_0}, \ldots, \overrightarrow{m_k})\right)\end{array} \middle| \; \mathsf{OIP}_\mathsf{out}\left(\rho, \mathsf{msg}_\mathsf{R}, \mathsf{msg}_\mathsf{S}\right) = \overrightarrow{m_0} + \textstyle\sum_{i\in[k]} b_i \overrightarrow{m_i}\right] = 1.$$

– ***Security against Semi-Honest Sender****: The following holds for any* $(b_1, \ldots, b_k) \in \mathcal{D}^k$ *and* $(b'_1, \ldots, b'_k) \in \mathcal{D}^k$*, where* $\exists i \in [k]$ *s.t.* $b_i \neq b'_i$

$$\left\{(\mathsf{msg}_\mathsf{R}, \rho) \leftarrow \mathsf{OIP}_\mathsf{R}\left(1^\lambda, (b_1, \ldots, b_k)\right) \;\middle|\; \mathsf{msg}_\mathsf{R}\right\} \approx_c \left\{(\mathsf{msg}'_\mathsf{R}, \rho') \leftarrow \mathsf{OIP}_\mathsf{R}\left(1^\lambda, (b'_1, \ldots, b'_k)\right) \;\middle|\; \mathsf{msg}_\mathsf{R}\right\}.$$

– ***Security against Semi-Honest Receiver****: For every PPT adversary* $\mathcal{A}$ *corrupting the receiver, there exists a PPT simulator* $\mathcal{S}_\mathsf{R}$ *such that for any choice of* $(b_1, \ldots, b_k) \in \mathcal{D}^k$ *and* $(\overrightarrow{m_0}, \ldots, \overrightarrow{m_k}) \in \mathcal{D}^{m \times (k+1)}$*, the following holds:*

$$\mathsf{OIP}_\mathsf{S}\left(1^\lambda, \mathsf{msg}_\mathsf{R}, (\overrightarrow{m_0}, \ldots, \overrightarrow{m_k})\right) \approx_c \mathcal{S}_\mathsf{R}(1^\lambda, \rho, \mathsf{msg}_\mathsf{R}, \overrightarrow{m_0} + \sum_{i \in [k]} b_i \overrightarrow{m_i}),$$

where $(\mathsf{msg}_\mathsf{R}, \rho) \leftarrow \mathsf{OIP}_\mathsf{R}(1^\lambda, (b_1, \ldots, b_k))$*.*

– ***Security against a Malicious Receiver****: For every PPT adversary* $\mathcal{A}$ *corrupting the receiver, there exists a PPT simulator* $\mathcal{S}_\mathsf{R} = (\mathcal{S}^1_\mathsf{R}, \mathcal{S}^2_\mathsf{R})$*, such that for any choice of* $(\overrightarrow{m_0}, \ldots, \overrightarrow{m_k}) \in \mathcal{D}^{m \times (k+1)}$*, the following holds:*

$$\left| \Pr\left[\mathsf{IDEAL}_{\mathcal{S}_\mathsf{R}, \mathcal{F}_\mathsf{OIP}}(1^\lambda, \overrightarrow{m_0}, \ldots, \overrightarrow{m_k}) = 1\right] - \Pr\left[\mathsf{REAL}_{\mathcal{A}, \mathsf{OIP}}(1^\lambda, \overrightarrow{m_0}, \ldots, \overrightarrow{m_k}) = 1\right] \right| \leq \tfrac{1}{2} + \mathsf{negl}(\lambda).$$

Where experiments $\mathsf{IDEAL}_{\mathcal{S}_\mathsf{R}, \mathcal{F}_\mathsf{OIP}}$ *and* $\mathsf{REAL}_{\mathcal{A}, \mathsf{OIP}}$ *are defined as follows:*

Exp $\mathsf{IDEAL}_{\mathcal{S}_\mathsf{R}, \mathcal{F}_\mathsf{OIP}}\left(1^\lambda, \overrightarrow{m_0}, \ldots, \overrightarrow{m_k}\right)$:

- $\mathsf{msg}_\mathsf{R} \leftarrow \mathcal{A}(1^\lambda)$
- $(b_1, \ldots, b_k) \leftarrow \mathcal{S}^1_\mathsf{R}(1^\lambda, \mathsf{msg}_\mathsf{R})$
- $\mathsf{out} \leftarrow \mathcal{F}_\mathsf{OIP}(\overrightarrow{m_0}, \ldots, \overrightarrow{m_k}, b_1, \ldots, b_k)$
- $\mathsf{msg}_\mathsf{S} \leftarrow \mathcal{S}^2_\mathsf{R}(1^\lambda, \mathsf{out}, \mathsf{msg}_\mathsf{R})$
- Output $\mathcal{A}(\mathsf{msg}_\mathsf{S})$

Exp $\mathsf{REAL}_{\mathcal{S}_\mathsf{R}, \mathcal{F}_\mathsf{OIP}}\left(1^\lambda, \overrightarrow{m_0}, \ldots, \overrightarrow{m_k}\right)$:

- $\mathsf{msg}_\mathsf{R} \leftarrow \mathcal{A}(1^\lambda)$
- $\mathsf{msg}_\mathsf{S} \leftarrow \mathsf{OIP}_\mathsf{S}\left(1^\lambda, \mathsf{msg}_\mathsf{R}, (\overrightarrow{m_0}, \ldots, \overrightarrow{m_k})\right)$
- Output $\mathcal{A}(\mathsf{msg}_\mathsf{S})$

We present a construction of such OIPs from linearly homomorphic encryption in the full-version of this paper. We show that if the underlying linearly homomorphic encryption has rate-1, then so does the OIP protocol.

4 MPC Interface

As discussed in the introduction, all of our compilers make use of an underlying secure computation protocol with certain properties. In this section, we describe the properties that we want from these underlying protocols.

We model these requirements as a reactive functionality (denoted as $\mathcal{F}_\mathsf{mpc}$). At a high level, we require secret sharing based MPC that evaluate a given circuit in a gate-by-gate manner and maintain an invariant that the parties hold

a secret sharing of the values induced on each intermediate wire in the circuit. A formal description of this reactive functionality appears in Figure 1.

For ease of notation, in our protocol descriptions, we shall let $[varid]$ denote the value stores by the functionality under $(varid, a)$; and we will write $[z] = [x] + [y]$ as a shorthand for calling **Add** and $[z] = [x] \cdot [y]$ as a shorthand for calling **Multiply**. And by abuse of notation, we will let $varid$ denote the value, x, of the data item held in location $(varid, x)$. We use $[x]_i$ to denote the share of x given to party P_i in the underlying MPC.

To the best of our knowledge, most secret sharing based protocols [33,25,12,16,13] securely implement this reactive functionality in the presence of a malicious adversary who can corrupt arbitrary number of parties. Moreover, most of these protocols are capable of evaluating circuits over any field/ring.

It is easy to see that any such secret sharing based MPC that evaluates the circuit in a gate-by-gate manner and maintains the invariant that parties hold shares of all intermediate wires in the circuit will trivially have support for the **Initialize Input, Initialize constant, Add, Add by const, Multiply, Multiply by const, Function** and **Output Private Shares** calls. Moreover, since the multiplication in these protocols typically requires parties to actually generate and compute shares of random values, the **Random** call is also implemented by these protocols. We now discuss how the remaining calls can be implemented in both the semi-honest and malicious settings.

Semi-Honest Setting. The only other calls used in our semi-honest protocols are **Random Bit** and **Output**. As observed in some of these protocols, **Random Bit** is also very easy to implement (especially in the semi-honest setting). This is done by requiring each party P_i to randomly sample $b_i \in \{1, -1\}$ and secret share it amongst all the parties. The parties then add all the shares obtained from all parties (let the resulting shares be $[s]$) and then compute $\frac{[s]+1}{2}$. The resulting shares will be of a random bit. **Share Zero** can be realized with semi-honest security by having every party secret share 0 and then requiring each party to locally sum up its shares. Finally, it is easy to see that the **Output** call can also be easily implemented, since the parties actually hold shares of all intermediate values. To reconstruct the output, they can simply broadcast their respective shares to all parties and then run the reconstruction algorithm.

Malicious Setting. While protocols such as SPDZ [16] and its descendants [33,25,12] (that use MACs w.r.t. a global key) delegate the check that ensures that these shares are indeed consistent with the "correct" values to the end of the protocol, we show that these protocols still securely implement all remaining calls in the $\mathcal{F}_{\mathsf{mpc}}$ functionality.

Intuitively, since these protocols delegate the malicious security/consistency checks to the end the protocol, the only place where we need to ensure that the shares held by the parties for any particular wire are indeed consistent and correct is when those values are reconstructed or are used outside of this MPC protocol, i.e., in the OIP and when the outcome of OIP is returned to the MPC. The subcalls inside $\mathcal{F}_{\mathsf{mpc}}$ that are really affected by this are **Initialize Input,**

Random, Share Zero, Check Zero and **Output Shares** and **Output**. As discussed above, **Initialize Input** and **Random** are already implemented by these protocols.

- **Check Zero:** For this sub-call, we observe that given authenticated additive shares $([x_1], [m_1])$, $([x_2], [m_2])$, with $m_1 = k * x_1$, $m_2 = k * x_2$ where k is the global MAC key, parties can compute $[m] = [m_1] - [m_2]$ locally, followed by having each player P_i first commit and then broadcast its share $[m]_i$ to reconstruct $[m]$ and check if $m = \sum_i m_i = 0$.
- **Share Zero:** For this we can augment the semi-honest **Share Zero** protocol described above with an asymptotically efficient batch-wise check to ensure malicious security. Specifically, to verify the outputs of the ℓ semi-honest **Share Zero** calls $[x_1], \ldots, [x_\ell]$, parties can publicly sample ℓ random values $\{r_i\}_{i=1}^{\ell}$ and compute a random linear combination $[r] = \sum_{i=1}^{\ell} r_i[x_i]$ followed by running the **Check Zero** call on $[r]$ and a trivial sharing of 0 (each party P_i's share is 0).
- **Output and Output Share:** As discussed above authenticated shares in the above protocols are of the form $([x], [m])$, where $m = k * x$ and k is the global MAC key. For both of these sub-calls, the parties first broadcast their shares $[x]$ and reconstruct. Then the parties can compute $x \cdot [k]$ and run **Check Zero** to check if the resulting shares reconstruct to the same value as the shares $[m]$. This is very similar to "MAC check" subprotocol already implemented in [33].

  We note that the above proposed protocols only reveal shares $[x]$ and not $[m]$. Indeed, revealing all shares of both x an m will trivially give away the global MAC key and make the protocol insecure. To make this compatible with our maliciously secure protocol, we assume that when the parties use the shares generated via $\mathcal{F}_{\mathsf{mpc}}$ outside of $\mathcal{F}_{\mathsf{mpc}}$ (i.e., to compute the OIP messages), they can do so on the "unauthenticated shares", i.e., on only the $[x]$ part and not on the $[m]$ part. Now, before, using the shares obtained as output of this OIP in $\mathcal{F}_{\mathsf{mpc}}$, we can make them "authenticated" by computing the corresponding $[m]$ shares for this output. This can be done trivially, since the parties hold a secret sharing of the global MAC key. This is a standard approach used in many of the above protocols including MASCOT [33].

  Moreover, we remark that the above proposed modification does not cause our compiler or the compiled protocols to be insecure in any way. This is because, the authentication mechanism used on the shares is only specific to $\mathcal{F}_{\mathsf{mpc}}$ and not to the primitives used outside of it. As a result, outside of $\mathcal{F}_{\mathsf{mpc}}$, an adversary can easily modify the authenticated shares in whatever way they want. Hence, in principle the following strategies are equivalent – (1) where the computations done outside of $\mathcal{F}_{\mathsf{mpc}}$ are performed on authenticated shares. (2) where the computations done outside of $\mathcal{F}_{\mathsf{mpc}}$ are performed on unauthenticated shares, but we authenticate the output of those computations before they are used in $\mathcal{F}_{\mathsf{mpc}}$ again.

Functionality $\mathcal{F}_{\mathsf{mpc}}$

Initialize Input: On input $(initinp, varid, P_i)$ from P_i (for each $i \in [n]$) with a fresh identifier $varid$ the functionality stores $(varid, [x])$.
Initialize constant: On input $(initconst, constid, c)$ from each P_i ($i \in [n]$) with a fresh identifier $varid$ the functionality stores $(const, c)$.
Random: On command $(rand, varid)$ from all parties, with a fresh identifier $varid$, the functionality selects a random value r, stores $(varid, [r])$ and sends the respective share $[r]_i$ to party P_i (for each $i \in [n]$)
Random Bit: On command $(bitrand, varid)$ from all parties, with a fresh identifier $varid$, the functionality selects a random bit $b \in \{0, 1\}$, stores $(varid, [b])$ and sends the respective share $[b]_i$ to party P_i (for each $i \in [n]$)
ShareZero: On command $(sharezero, varid)$ from all parties, with a fresh identifier $varid$, the functionality computes, stores $(varid, [0])$ and sends the respective share $[0]_i$ to party P_i (for each $i \in [n]$).
Add: On command $(add, varid_1, varid_2, varid_3)$ from all parties (if $varid_1$, $varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, [x]), (varid_2, [y])$ and stores $(varid_3, [x + y])$.
Add by const: On command $(add, constid_1, varid_2, varid_3)$ from all parties (if $constid_1$, $varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(constid_1, c), (varid_2, [x])$ and stores $(varid_3, [c + x])$.
Multiply: On input $(mult, varid_1, varid_2, varid_3)$ from all parties (if $varid_1$, $varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, [x]), (varid_2, [y])$ and stores $(varid_3, [x \cdot y])$.
Multiply by const: On command $(mult, constid_1, varid_2, varid_3)$ from all parties (if $constid_1$, $varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(constid_1, c), (varid_2, [x])$ and stores $(varid_3, [c \cdot x])$.
Function: On input $(func, f, varid_1, \ldots, varid_n, varid_{\mathsf{out}})$ from all parties, the functionality retrieves $(varid_1, [x_1]), \ldots, (varid_n, [x_n])$ and stores $(varid_{\mathsf{out}}, [f(x_1, \ldots, x_n)])$.
Output Shares: On input $(outshare, varid)$ from all parties, the functionality retrieves $(varid, [x])$ and outputs all shares $[x]$ to all parties.
Output Private Shares: On input $(outprivshare, varid)$ from all parties, the functionality retrieves $(varid, [x])$ and outputs the respective share $[x]_i$ to party P_i (for each $i \in [n]$).
Check Zero: On input $(fcheckzero, varid_1, varid_2)$ from all parties, the functionality retrieves $(varid_1, [x_1]), (varid_2, [x_2])$ and outputs 1 w.h.p if $x_1 = x_2$ and otherwise it outputs 0 and aborts.
Output: On input $(out, varid)$ from all honest parties (if $varid$ is present in memory), the functionality retrieves $(varid, [x])$ and outputs x to all players.

Fig. 1: A Required Ideal Functionality for MPC

5 Non-Constant Round Semi-Honest Branching MPC

In this section, we present our semi-honest compiler for distributed computation of a circuit with conditional branches.

Let the circuit/function be such that it consists of an initial sub-function f_1, followed by the k branches and then a sub-function f_2. We assume that the parties have access to $\mathcal{F}_{\mathsf{mpc}}$ (see Figure 1). When evaluated using $\mathcal{F}_{\mathsf{mpc}}$, the output of f_1 is a secret sharing of the inputs to the branching part and a secret sharing of the unary representation of the index associated with the branch that needs to be executed (henceforth referred to as the active branch). The output of the branching part is a secret sharing of the inputs to the function f_2.

Given a circuit C, we assume that the parties decide on some canonical ordering of the gates in the circuit, such that gate i only takes as inputs the values output by the gates $j < i$. We assume w.l.o.g. that the i^{th} gate in C has fan-in 2

and the outgoing wire of any gate can act as the incoming wire for any number of gates.[4]

For simplicity, we assume that all branches are of the same size and have G gates. Our protocol can be easily extended to the scenario where the branches are of varying sizes by suitably padding the smaller branches with fake gates. Let ℓ be the length of inputs to the branching part of the function. For evaluating this part, we assume that there are ℓ input gates that are common to all branches. We set both the incoming and outgoing labels for the wires coming out of these gates as $1, \ldots, \ell$ respectively. For each branch $m \in [k]$, and each gate i in this branch, we assign outgoing label $i + \ell$ to the wire coming out of this gate and incoming labels $\ell + 2i - 1$ and $\ell + 2i$ respectively to its two incoming wires. Therefore, we assume that the number of unique outgoing labels assigned in a branch are $G + \ell$, while the total number of unique incoming labels assigned in a branch are $W = 2G + \ell$. We present a slightly optimized version of the protocol described in the introduction, namely that only requires parties to sample 1 mask per wire, instead of 2 masks.

Let π be the mapping corresponding to a circuit C that maps incoming labels to the outgoing labels of each wire in C. For instance, $\pi(i)$ corresponds to the outgoing label of the wire with incoming label i. Let $\mathsf{C}_1, \ldots, \mathsf{C}_k$ be the circuit representations of the k branches and let $\{\pi_1, \ldots, \pi_k\}$ be the corresponding mappings associated with these branches. Finally, we assume that the circuits and inputs are defined over some field $\mathbb{F}$.

Protocol. The parties start by invoking $(func, f_1, x_1, \ldots, x_n, \mathsf{inp}_1, \ldots, \mathsf{inp}_\ell, b_1, \ldots, b_k)$ in $\mathcal{F}_{\mathsf{mpc}}$ on their original inputs $x_1, \ldots, x_n$, to obtain shares of inputs to the branching part $[\mathsf{inp}_1], \ldots, [\mathsf{inp}_\ell]$, where $|\ell|$ is the total input length and shares $[b_1], \ldots, [b_k]$, where $b_1 \ldots b_k$ is the unary representation of the index associated with the active branch. Given these shares, parties run the protocol presented in Figure 2. The output of this protocol is a secret sharing of the inputs to f_2 (i.e., the last part of the circuit). Let m be the length of these inputs. The parties finally invoke $(func, f_2, y_1, \ldots, y_m, \mathsf{out})$ and (out, out) in $\mathcal{F}_{\mathsf{mpc}}$ to learn the final output out.

Optimization. A naive implementation of the online phase in the above protocol will result in a round complexity that depends on the maximum number of gates in any particular branch. This can be improved to be proportional to the maximum multiplicative depth of any branch by using a simple optimization. For simplicity, lets assume that all branches have the same depth and each layer of each branch contains the same number of gates. We know that the gates on level ℓ only depend on the outgoing wires of gates on layers $< \ell$. We can therefore evaluate all the gates in a particular level in parallel. This simple idea can also be extended to the case where the branches have different depths and widths. In

[4] Our compiler can work with circuits that have gates with arbitrary fan-out. In our construction, it suffices to view such gates as having a single outgoing wire that acts as the incoming wire for multiple gates. Hence, we only assign a single label to the outgoing wire of each gate.

Semi-Honest Protocol

The protocol is described in the $\mathcal{F}_{\mathsf{mpc}}$-hybrid model. Parties have shares of inputs to the branches, i.e., $[\mathsf{inp}_1], \ldots, [\mathsf{inp}_\ell]$ and shares of a unary representation of the active branch, i.e., $[b_1], \ldots, [b_k]$.

– **Pre-processing Phase:**
  1. **Sample masks:** For each input and gate $g \in [\ell+G]$, parties invoke $(rand, \mathsf{mask}_g)$ in $\mathcal{F}_{\mathsf{mpc}}$ to obtain shares $[\mathsf{mask}_g]$. For each branch $m \in [k]$, let $[\overrightarrow{\mathsf{mask}_{\pi_m}}] = [\mathsf{mask}_{\pi_m(1)}] \| \ldots \| [\mathsf{mask}_{\pi_m(W)}]$.
  2. **Shares of zeros:** For each $w \in [W]$ and $i \in [n]$, parties invoke $(sharezero, X_{w,i})$ in $\mathcal{F}_{\mathsf{mpc}}$ to get shares $[X_{w,i}]$, where $X_{w,i} = 0$. For each $i \in [n]$, let $[\overrightarrow{X_i}] = [X_{1,i}] \| \ldots \| [X_{W,i}]$.
  3. **Pairwise OIP:** Each pair of parties P_R and P_S ($\forall \mathsf{R}, \mathsf{S} \in [n]$) engage in a two-message semi-honest OIP as follows, where P_R acts as the receiver and P_S acts as the sender:
     - **Receiver:** P_R computes $(\rho, \mathsf{msg}_\mathsf{R}) \leftarrow \mathsf{OIP}_\mathsf{R}(1^\lambda, [b_1]_\mathsf{R}, \ldots, [b_k]_\mathsf{R})$ and sends msg_R to P_S.
     - **Sender:** P_S computes $\mathsf{msg}_\mathsf{S} \leftarrow \mathsf{OIP}_\mathsf{S}(1^\lambda, \mathsf{msg}_\mathsf{R}, [\overrightarrow{X_\mathsf{R}}]_\mathsf{S}, [\overrightarrow{\mathsf{mask}_{\pi_1}}]_\mathsf{S}, \ldots, [\overrightarrow{\mathsf{mask}_{\pi_k}}]_\mathsf{S})$ and sends msg_S to P_R.
     - **Output:** P_R computes $\overrightarrow{\mathsf{share}_{\mathsf{R},\mathsf{S}}} \leftarrow \mathsf{OIP}_{\mathsf{out}}(\rho, \mathsf{msg}_\mathsf{R}, \mathsf{msg}_\mathsf{S})$.
  4. **Δ values:** Each party P_i (for $i \in [n]$) computes $[\overrightarrow{\Delta}]_i = \sum_{j \in [n]} \overrightarrow{\mathsf{share}_{j,i}}$, where $[\overrightarrow{\Delta}] = [\Delta_1] \| \ldots \| [\Delta_W]$.

– **Online Phase :**
  1. **Inputs:** For each input wire $i \in [\ell]$, parties compute $[u_i] = [\mathsf{inp}_i] + [\mathsf{mask}_i]$. and invoke (out, u_i) in $\mathcal{F}_{\mathsf{mpc}}$ to obtain u_i in the clear.
  2. **Circuit Evaluation:** For each gate $g \in [G]$, let $\mathsf{left} = \ell+2g-1$ and $\mathsf{right} = \ell+2g$ be the incoming wire labels of its input wires. Let $\mathsf{type}_{m,g}$ be the gate type for gate g in C_m ($\forall m \in [k]$), where $\mathsf{type}_{m,g} = 0$ denotes an addition gate and $\mathsf{type}_{m,g} = 1$ denotes a multiplication gate. Parties compute the following using $\mathcal{F}_{\mathsf{mpc}}$:
     (a) For $w \in \{\mathsf{left}, \mathsf{right}\}$, compute $[y_w] = \sum_{m=1}^{k} \left(u_{\pi_m(w)} \cdot [b_m]\right) + [\Delta_w]$.
     (b) Compute $[\mathsf{type}_g] = \sum_{m=1}^{k} \left(\mathsf{type}_{m,g} \cdot [b_m]\right)$
     (c) Compute $[z_g] = (1 - [\mathsf{type}_g])([y_{\mathsf{left}}] + [y_{\mathsf{right}}]) + [\mathsf{type}_g]([y_{\mathsf{left}}] \cdot [y_{\mathsf{right}}])$.
     (d) Compute $[u_{\ell+g}] = [z_g] + [\mathsf{mask}_{\ell+g}]$ and invoke (out, u_s) in $\mathcal{F}_{\mathsf{mpc}}$ to obtain u_s in the clear.
  3. **Output:** For each output gate g, compute $[z_g] = \sum_{m=1}^{k} \left(u_{\pi_m(w)} \cdot [b_m]\right) + [\Delta_w]$.

Fig. 2: Semi-Honest Compiler

that case, let x_ℓ (and y_ℓ resp.) be the minimum (and maximum resp.) number of gates on level ℓ in any branch. We can evaluate the first x_ℓ gates in parallel. Then in the next round we can evaluate the $y_\ell - x_\ell + x_{\ell+1}$ gates in parallel. This ensures that the overall round complexity of the online phase will only depend on the depth of the branches.

Complexity Analysis. We now analyze the communication complexity of the above semi-honest protocol. If we use a rate-1 OIP, the communication complexity in the pre-processing phase is $O(n^2|C_{\max}|+n^2k\lambda)$, where $|C_{\max}|$ is the size of the largest branch. In the online phase for each gate we perform both addition and multiplication and then choose between the two. As a result we perform 2 multiplications per gate. The communication complexity of the online phase is $O(2 \times \mathtt{CC}(|C_{\max}|))$, where $\mathtt{CC}(|C_{\max}|)$ is the communication complexity incurred upon evaluating $C_{\max}$ using the underlying MPC.

Overall, given the above protocol and optimizations, we obtain the following result. Due to space constraints, we defer the security proof of this construction to the full-version of this paper.

Theorem 1. *Let λ be the security parameter and $\mathcal{F}$ be a function class consisting of functions of the form $f(\overrightarrow{x}) = f_2(f_{\mathsf{br}}(f_1(\overrightarrow{x})))$, where $f_{\mathsf{br}} \coloneqq \{g_1, \ldots, g_k\}$ is a function consisting of k conditional branches, defined as $f_{\mathsf{br}}(i, \overrightarrow{x}) = g_i(\overrightarrow{x})$. Assuming the existence of a rate-1 two-message semi-honest secure OIP (see Definition 1), there exists an MPC protocol in the $\mathcal{F}_{\mathsf{mpc}}$-hybrid model (see Section 4) for computing any $f \in \mathcal{F}$ that achieves semi-honest security against an arbitrary number of corruptions and incurs a communication overhead of $O(n^2(k\lambda + |C_{\mathsf{max}}|))$.*

In the full-version of this paper, we show that a rate-1 two-message semi-honest secure OIP can be constructed from rate-1 linearly homomorphic encryption. Such encryptions are known [17,8,15,38] from a variety of assumptions including LWE, Ring LWE and DDH assumption.

6 Constant Round Semi-Honest Branching MPC

In this section we present our constant round semi-honest protocol for distributed computation of a branching circuit.

As discussed in the technical overview, we instantiate a random function based on the RLWE assumption for our protocol that works as an approximate key homomorphic PRF. We briefly recall the variant of the decisional RLWE hardness assumption stated by Ben-Efraim et al. [4]. Let $p = 2N + 1$ be a prime, where N, called the dimension or security parameter, is a power of 2. Let $\mathcal{R}_p = \mathbb{Z}_p[X]/(X^N + 1)$ be the polynomial ring over $\mathbb{Z}_p$ modulo $X^N + 1$. We start by recalling the decisional RLWE assumption.

Definition 2 (Decisional Ring LWE Problem). *Any non-uniform* PPT *adversary cannot distinguish between $\{(a_i, b_i)\}_{i\in I}$ and $\{(a_i, a_i \cdot k + \delta_i)\}_{i\in I}$ with non-negligible probability where $\{a_i\}_{i\in I}$, $\{b_i\}_{i\in I}$ and k are chosen uniformly at random from $\mathcal{R}_p$ and the coeffecients of $\{e_i\}_{i\in I}$ are sampled from χ, a spherical Gaussian distribution.*

By transforming to the Hermite normal form, the RLWE assumption also holds if k is chosen from a spherical Gaussian distribution. In general, it is also necessary to bound the number of samples $|I|$; say $|I| = O(1)$ or $|I| = O(\log N)$.

Our protocol follows the BMR approach which involves sampling a pair of keys k_w^0, k_w^1 for each wire w in the circuit. A garbled table is then constructed for each gate such that the key corresponding to the value on the output wire is encrypted using the keys corresponding to the input values. Since the position of each ciphertext in the garbled table leaks information about its plaintext, a private random mask bit $\gamma_w \in \{0,1\}$ is sampled for each wire w and the masks for the input wires are used to permute the rows of the garbled table for each gate. Let the external value β_w on a wire be the plaintext value ρ_w on the wire masked with the mask γ_w i.e., $\beta_w = \rho_w \oplus \gamma_w$. Then, the masks on the input wires are used to permute the rows of the garbled table such that the external values on the input wires can be used to index into the required row of the garbled table. Thus, to ensure that parties decrypt the correct row when evaluating the circuit, the mask for the output wire has to also be included in the ciphertext for each row. We use the approach of Ben-Efraim et al. [4], where the last coordinate of the keys k_w^0, k_w^1 for each wire are set to 0, which slightly reduces security, and the external value is embedded into this coordinate during encryption. We use $k \| e$ to denote that the bit e was embedded in the last coordinate of the key k.

As observed in [4], since the plaintext and key come from the same set when computing the ciphertexts for the garbled table, we sample coefficients for both the keys and errors from Gaussian distribution χ, similar to the RLWE errors to ensure that decryption is possible. Moreover, Ben-Efraim et al. [4] also show that overall, it suffices to use just $8 \cdot f_{\mathsf{out}}$ distinct public random elements of the form $A_g^{u,v}$ from the ring, where f_{out} is the maximal fan-out of the circuit.

The garbling phase is presented in Figure 3 and the evaluation phase is presented in Figure 4. We adopt similar notation to the semi-honest protocol presented in Figure 2 and use incoming and outgoing labels for each wire. Let ℓ be the number of input wires to the branching part of the function, we set the incoming and outgoing labels for these wires to be $1, \ldots, \ell$. For gate g in each branch, we set the outgoing wire label to be $\ell + g$, the left incoming wire label to be $\ell + 2g - 1$ and the right incoming wire label to be $\ell + 2g$. We also let π_m for each $m \in [k]$ to be the mapping that maps incoming labels to the outgoing labels of each wire for the m-th branch.

Finally, we remark that we require the underlying MPC protocol that securely realizes $\mathcal{F}_{\mathsf{mpc}}$ to run in constant number of rounds for constant depth circuits. This is to ensure that our protocol has constant number of rounds. This is true for most secret sharing based protocols that evaluate the circuit in a gate-by-gate manner.

Complexity Analysis. We assume that the size of the ring $\mathcal{R}_p$ is in $O(\lambda)$. If we use a rate-1 semi-honest secure OIP, the communication complexity in the garbling phase is $O(n^2|C_{\max}| + n^2 k \lambda \mathtt{CC}(\lambda|C_{\max}|))$, where $|C_{\max}|$ is the size of the largest branch and $\mathtt{CC}(\lambda|C_{\max}|)$ is the communication complexity incurred upon evaluating $C_{\max}$ using the underlying MPC. In the evaluation phase, the communication cost incurred is for reconstructing $O(\lambda|C_{\max}|)$ shares corresponding to the garbling material.

Garbling Phase of the Constant Round Semi-Honest Protocol

The protocol is described in the $\mathcal{F}_{\mathsf{mpc}}$-hybrid model which computes over $\mathcal{R}_p$. The parties have shares of a unary representation of the active branch, i.e., $[b_1], \ldots, [b_k]$. For each gate $g \in [G]$, let $\mathsf{left}_g = \ell + 2g - 1$ and $\mathsf{right}_g = \ell + 2g$ be the incoming wire labels of its input wires and let $\mathsf{out}_g = \ell + g$ be the outgoing wire.

1. **Sample masks:** For each input and gate $g \in [\ell + G]$, the parties invoke $(randbit, \gamma_g)$ in $\mathcal{F}_{\mathsf{mpc}}$ to obtain shares $[\gamma_g]$. For each branch $m \in [k]$, let $[\overrightarrow{\gamma_{\pi_m}}] = [\gamma_{\pi_m(1)}] \| \ldots \| [\gamma_{\pi_m(W)}]$.
2. **Sample keys:** For each $g \in [\ell + G]$, and $j \in \{0, 1\}$ each party P_i (for $i \in [n]$) locally samples its share $[k_g^j]_i \leftarrow \chi^N$ and sets the last coordinate of its share to 0.
3. **Compute LWE expansions:** For each $u, v \in \{0, 1\}$, $g \in [G]$ each party P_i (for $i \in [n]$) locally samples $\delta_{m,g}^{u,v,i} \leftarrow \chi^N$ and computes $[\psi_{m,g}^{u,v}]_i = A_g^{u,v} \cdot ([k_{\pi_m(\mathsf{left}_g)}]_i + [k_{\pi_m(\mathsf{right}_g)}]_i) + \delta_{m,g}^{u,v,i}$. Let $[\overrightarrow{\psi_m}] = [\psi_{m,1}^{0,0}] \| [\psi_{m,1}^{0,1}] \| [\psi_{m,1}^{1,0}] \| [\psi_{m,1}^{1,1}] \| \ldots \| [\psi_{m,G}^{0,0}] \| [\psi_{m,G}^{0,1}] \| [\psi_{m,G}^{1,0}] \| [\psi_{m,G}^{1,1}]$.
4. **Shares of zero:** For each $i \in [n]$ and $j \in [W + 4G]$, the parties invoke $(sharezero, X_{j,i})$ in $\mathcal{F}_{\mathsf{mpc}}$ to get shares $[X_{j,i}]$, where $X_{j,i} = 0$. For each $i \in [n]$, let $[\overrightarrow{X_i}] = [X_{1,i}] \| \ldots \| [X_{W+4G,i}]$.
5. **Pairwise OIP:** Each pair of parties P_{R} and P_{S} ($\forall \mathsf{R}, \mathsf{S} \in [n]$) engage in a two-message semi-honest OIP as follows, where P_{R} acts as the receiver and P_{S} acts as the sender:
   - **Receiver:** P_{R} computes $(\rho, \mathsf{msg}_{\mathsf{R}}) \leftarrow \mathsf{OIP}_{\mathsf{R}}(1^\lambda, [b_1]_{\mathsf{R}}, \ldots, [b_k]_{\mathsf{R}})$ and sends $\mathsf{msg}_{\mathsf{R}}$ to P_{S}.
   - **Sender:** For each $m \in [1, k]$ let $[\overrightarrow{x_m}] = [\overrightarrow{\gamma_{\pi_m}}] \| [\overrightarrow{\psi_m}]$. P_{S} computes $\mathsf{msg}_{\mathsf{S}} \leftarrow \mathsf{OIP}_{\mathsf{S}}(1^\lambda, \mathsf{msg}_{\mathsf{R}}, [\overrightarrow{X_{\mathsf{R}}}]_{\mathsf{S}}, [\overrightarrow{x_1}]_{\mathsf{S}}, \ldots, [\overrightarrow{x_k}]_{\mathsf{S}})$ and sends $\mathsf{msg}_{\mathsf{S}}$ to P_{R}.
   - **Output:** P_{R} computes $\overrightarrow{\mathsf{share}_{\mathsf{R},\mathsf{S}}} \leftarrow \mathsf{OIP}_{\mathsf{out}}(\rho, \mathsf{msg}_{\mathsf{R}}, \mathsf{msg}_{\mathsf{S}})$.

   For each $i \in [n]$, P_i computes $[\overrightarrow{\Gamma}] \| [\overrightarrow{\Psi}] = \sum_{j \in [n]} \overrightarrow{\mathsf{share}_{j,i}}$ where $\overrightarrow{\Gamma} = \Gamma_1 \| \ldots \| \Gamma_W$ and $\overrightarrow{\Psi} = \Psi_1^{0,0} \| \ldots \| \Psi_G^{1,1}$.
6. **Garble active branch:** Let $\mathsf{type}_{m,g}$ be the gate type for gate g in C_m ($\forall m \in [k]$), where $\mathsf{type}_{m,g} = 0$ denotes an XOR gate and $\mathsf{type}_{m,g} = 1$ denotes an AND gate. Parties do the following for each $g \in [G]$
   (a) Compute $[\mathsf{type}_g] = \sum_{i=m}^{k} \left(\mathsf{type}_{m,g} \cdot [b_m]\right)$.
   (b) For each $u, v \in \{0, 1\}$ let $e_{u,v,g}^{\mathrm{xor}} = u \oplus \Gamma_{\mathsf{left}_g} \oplus v \oplus \Gamma_{\mathsf{right}_g} \oplus \gamma_{\mathsf{out}_g}$, $e_{u,v,g}^{\mathrm{and}} = ((u \oplus \Gamma_{\mathsf{left}_g}) \wedge (v \oplus \Gamma_{\mathsf{right}_g})) \oplus \gamma_{\mathsf{out}_g}$, $e_g^{u,v} = \mathsf{type}_g(e_{u,v,g}^{\mathrm{and}} - e_{u,v,g}^{\mathrm{xor}}) + e_{u,v,g}^{\mathrm{xor}}$ and $K_g^{u,v} = e_g^{u,v}(k_{\mathsf{out}_g}^1 - k_{\mathsf{out}_g}^0) + k_{\mathsf{out}_g}^0$. For each $u, v \in \{0, 1\}$, compute $[K_g^{u,v} \| e_g^{u,v}]$ using $\mathcal{F}_{\mathsf{mpc}}$.
   (c) For each $u, v \in \{0, 1\}$ compute $[C_g^{u,v}] = [\Psi_g^{u,v}] + \lceil \sqrt{p} \rceil [K_g^{u,v} \| e_g^{u,v}]$.

Fig. 3: Garbling phase of the constant round (semi-honest) protocol

Overall, given the above protocol and optimizations, we obtain the following result. Due to space constraints, we defer the security proof of this construction to the full-version of this paper.

Evaluation Phase of the Constant Round Semi-Honest Protocol

The protocol is described in the $\mathcal{F}_{\mathsf{mpc}}$-hybrid model. The parties have shares of the inputs to the branches, i.e., $[\mathsf{inp}_1], \ldots, [\mathsf{inp}_\ell]$ and shares of a unary representation of the active branch, i.e., $[b_1], \ldots, [b_k]$.

1. For each input wire $w \in [\ell]$ parties compute $[\beta_w] = [\mathsf{inp}_w] \oplus [\gamma_w]$ and invoke $(out, [\beta_w])$ in $\mathcal{F}_{\mathsf{mpc}}$ to obtain β_w. For each $w \in [\ell]$, let $\beta_{1,w} = \ldots = \beta_{k,w} = \beta_w$.
2. For each input wire $w \in [\ell]$ parties invoke $(out, [k_w^{\beta_w}])$ in $\mathcal{F}_{\mathsf{mpc}}$ to obtain $k_w^{\beta_w}$. For each $w \in [\ell]$, let $K_{1,w}^{\beta_w} = \ldots = K_{k,w}^{\beta_w} = k_w^{\beta_w}$.
3. For each $u, v \in \{0,1\}$ and $g \in [G]$ parties invoke $(out, [C_g^{u,v}])$ in $\mathcal{F}_{\mathsf{mpc}}$ to obtain $C_g^{u,v}$.
4. For each $m \in [k]$ and $g \in [G]$, parties compute $C_g^{u,v} - A_g^{u,v} \cdot \left(K^u_{m,\pi_m(\mathsf{left}_g)} + K^v_{m,\pi_m(\mathsf{right}_g)}\right)$, where $u = \beta_{m,\pi_m(\mathsf{left}_g)}$ and $v = \beta_{m,\pi_m(\mathsf{right}_g)}$, and divide it by $\lceil\sqrt{p}\rceil$ to remove the error and recover $K_{m,\mathsf{out}_g}^{\beta_{m,\mathsf{out}_g}} \| \beta_{m,\mathsf{out}_g}$.
5. For each output gate g, parties compute $[z_g] = \sum_{m=1}^{k} \beta_{m,\mathsf{out}_g} [b_m] \oplus [\gamma_g]$ using $\mathcal{F}_{\mathsf{mpc}}$.

Fig. 4: Evaluation phase of the constant round (semi-honest) protocol

Theorem 2. *Let λ be the security parameter and $\mathcal{F}$ be a function class consisting of functions of the form $f(\overrightarrow{x}) = f_2(f_{\mathsf{br}}(f_1(\overrightarrow{x})))$, where $f_{\mathsf{br}} \coloneqq \{g_1, \ldots, g_k\}$ is a function consisting of k conditional branches, defined as $f_{\mathsf{br}}(i, \overrightarrow{x}) = g_i(\overrightarrow{x})$. Assuming that a rate-1 two-message semi-honest secure OIP exists (see Definition 1) and that the decisional RLWE problem holds (see Definition 2), there exists a constant-round MPC protocol in the $\mathcal{F}_{\mathsf{mpc}}$-hybrid model (see Section 4) for computing any $f \in \mathcal{F}$ that achieves semi-honest security against an arbitrary number of corruptions and incurs a communication overhead of $O(n^2\lambda(k + |C_{\mathsf{max}}|))$.*

Note that if we instatiate the rate-1 two-message semi-honest secure OIP using a rate-1 RLWE-based linearly homomorphic encryption, then the above theorem yields a protocol that only relies on the hardness of decisional RLWE.

7 Implementation

We implement and benchmark our semi-honest non-constant round protocol from Section 5. The code is publicly available at `https://github.com/rot256/research-branching-mpc`. In addition to the code and instructions used for benchmarking, the repository also contains the raw data used in this paper and scripts used to create the plots.

7.1 How We Benchmark

Underlaying MPC. We implement tour semi-honest compiler on top of two different multi-party computation protocols.

1. *Quadratic Dependence on the Number of Parties.* A semi-honest variant of MASCOT [33] (MASCOT without sacrificing and message authentication codes) over the prime field $\mathbb{F}_{2^{16}+1} = \mathbb{Z}/(\texttt{0x10001}\ \mathbb{Z})$ provided by MP-SPDZ [32] (called "`semi-party.x`"). We simply invoke the MP-SPDZ implementation as a black-box: wrapping each instance of "`semi-party.x`" in a program which provides provides inputs/outputs to the party. Since MP-SPDZ povides a universal interface our implementation is agnostic with regards to the underlying MPC implementation: any reactive MPC in MP-SPDZ which allows computation over $\mathbb{F}_{2^{16}+1}$ could be swapped in with ease.
2. *Linear Dependence on the Number of Parties.* A batched semi-honest version of CDN [13] where we instantiate the linearly homomorphic encryption using the same ring LWE parameters described above. We implement this ourselves again using the Lattigo (more information below) library for the RLWE components.

CDN Implementation. We implement a semi-honest batched version of CDN, instantiating the linearly homomorphic encryption using the same parameters described above (the same as the OIP). To reduce the overhead (computational/communication) induced by the homomorphic encryption we execute multiplications in batches of 2^{12} (the dimension of the ring used for RLWE), by packing 2^{12} independent shares (over `0x10001`) into a single ciphertext and execute the CDN multiplication protocol on these in parallel. The decryption threshold is the full set of parties. The CDN implement is included in the same repository. To the best of our knowledge, this is the first known implementation of CDN.

Instantiating OIP and Ring LWE Parameters. In our implementation, we use an optimized version of OIP. We observe that the $O(n^2)$ overhead incurred from the use of pairwise-OIPs can be driven down to $O(n)$, if instead of a regular linearly homomorphic encryption, we use a threshold linearly homomorphic encryption (TLHE). TLHE are linearly homomorphic encryptions that comprise of a single public-key and where each party holds a "share" of the secret key. This share of the secret key can be used by the parties to decrypt to a share of the plaintext. As shown in [10], the keys for RLWE based threshold linearly homomorphic encryption can be setup very efficiently by the parties in a couple of rounds. At a high level, this observation allows us to reuse the sender and receiver messages of each party across multiple OIP instantiations and as a result, overall, each party only needs to send one receiver message and one sender message.

Recall that in our semi-honest protocol, the receiver and sender messages in all OIP instances are computed using the same shares of the index associated with the active branch and the masks. Each party can compute its receiver message by encrypting its shares of $b_1, \ldots, b_k$. Similarly, for the sender message, each party can compute an inner-product of these encryptions received from all parties and its shares of the permuted masks. Finally, all parties can add all the sender messages (which are also ciphertexts) received from all parties. This gives them an encryption of the permuted masks for the active branch. Now each

party can run threshold decryption using its share of the secret-key to obtain a sharing of the resulting inner-product.

We use BFV [17] over a cyclotomic ring of index 2^{13} and dimension 2^{12}, i.e. $R[X]/(X^{2^{12}}+1)$ where: $Q_1 := \texttt{0x7ffffec001}, Q_2 := \texttt{0x8000016001}, P := \texttt{0x40002001}$, $N := Q_1Q_2P, R := \mathbb{Z}/(N\mathbb{Z})$. This gives us a linearly homomorphic encryption scheme for vectors $\overrightarrow{v} \in (\mathbb{F}_{2^{16}+1})^{2^{12}}$, which additionally allows (full) threshold decryption. We use the `Lattigo` [1] library to implement all the RLWE components.

Benchmarking Platform. All benchmarks were run on a laptop with an Intel i7-11800H CPU (@ 2.3 GHz) and 64 GB of RAM. All networking is over the loopback interface and network latency was simulated using traffic control (`tc`) on Linux. We also do not restrict the bandwidth when comparing running times – note that this constitutes a relative "worst-case scenario" for our results: as our technique reduces communication, the relative performance gain for many branches would only increase by restricting bandwidth.

How The Branches Were Generated. During our benchmark each branch contained 2^{16} uniformly random gates: each gate is a multiplication/addition gate with probability $1/2$. We benchmark using "layered circuits", meaning each level contains 2^{12} gates which can be evaluated in parallel (to reduce the number of rounds). Subject to the layering constraint, the wiring is otherwise random: the inputs to each gate are sampled uniformly at random from all previous outputs (not just those in the last layer). We believe this distribution over circuits form a realistic benchmark for the expected performance across many real-world applications.

Averaging. We run all benchmarks 10 times and take the average.

7.2 Comparison of Communication Complexity

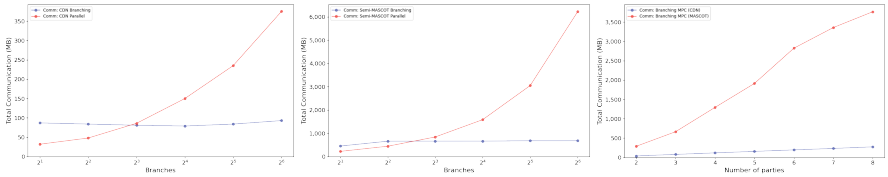
In Fig. 5, we compare the communication complexity of our technique to the naïve baseline solution of evaluating each branch in parallel using the underlying MPC. For the baseline solution we do not consider the additional overhead of multiplexing the output, i.e., selecting the output of the active branch.

Looking at Fig. 5 (a)/(b), we observe that our technique improves communication over the baseline for both CDN and MASCOT with 3 parties when the number of branches is ≥ 8. For less than 8 branches the communication overhead of the RLWE-based OIP and the need to evaluate universal gates (requiring the base-MPC to compute 3 multiplications) outweighs the communication saving of only executing the active branch. Upon reflection 8 branches is about the lowest number of branches we could hope to see savings for: recall that each branch contains $\approx 2^{15}$ multiplications[5], therefore the parallel execution of 6 branches requires the same number of multiplications as that of the 2^{16} universal gates

[5] Since the type of each gate in each branch is sampled uniformly at random.

used in our technique. As expected we also observe that the communication of our technique remains (nearly[6]) constant for any number of branches.

Lastly we fix the number of branches to 16 and plot (in Fig. 5 (c)) the communication complexity of our technique for a varying number of parties, as expected the communication of our compiler applied to MASCOT increases quadratically, while our technique preserves the linearly increasing communication of CDN; constant per-party communication (and computation).



(a) Branching MPC vs Parallel Evaluation of Branches with CDN. 3 Parties

(b) Branching MPC vs Parallel Evaluation of Branches with MASCOT. 3 Parties.

(c) Branching MPC, Communication for Different Number of Parties. Using 16 Branches.

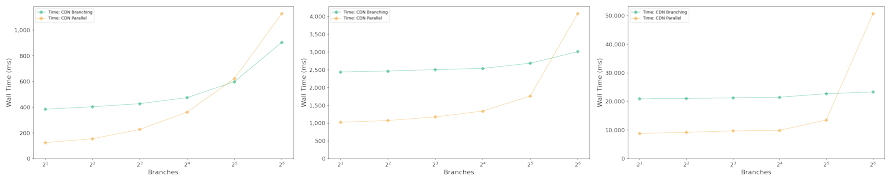
Fig. 5: Communication Complexity of Branching MPC compared to the base-line of evaluating each branch in parallel.

7.3 Comparison of Running Time

From Fig. 6 and Fig. 7, we observe that for sufficiently many branches our technique also reduces running time over the baseline for both CDN and semi-honest MASCOT. This is also expected: after the relatively high constant overhead of our technique, the marginal cost of adding another branch (of length ℓ) is that of: (1) $O(\ell)$ linear operations in the underlying MPC. (2) $O(\ell)$ $\langle ciphertext \rangle \times \langle plaintext \rangle$ operations in the RLWE based homomorphic encryption scheme. (3) $O(\ell)$ $\langle ciphertext \rangle + \langle ciphertext \rangle$ operations in the RLWE based homomorphic encryption scheme.

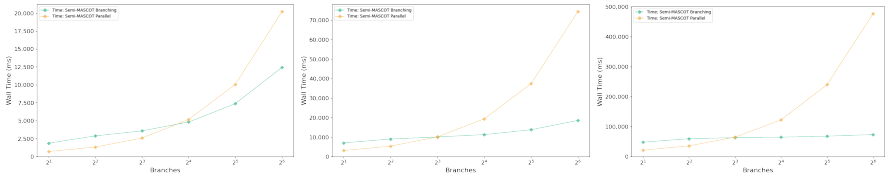
The first one introduces a very small cost (essentially that of reading the branch), the second is dominated by the cost of doing a number theoretic transform (NTT) on the plaintext (the players local share), which again is essentially that of computing a small constant number of fixed-size FFTs. We note that the NTTs are computed on random shares and could be relegated to a pre-computation phase. The final ciphertext/ciphertext addition is just a constant number of entry-wise additions of vectors in a small prime field – the cost of which is miniscule. Looking at Fig. 6 and Fig. 7 we observe that this marginal computational cost (of doing NTTs) has a higher influence when the network latency is low and quickly becomes insignificant as the latency increases.

[6] It grows slightly, since the unary representation of the selection wire must be shared/computed. However the computation of the branch completely dominates the communication.



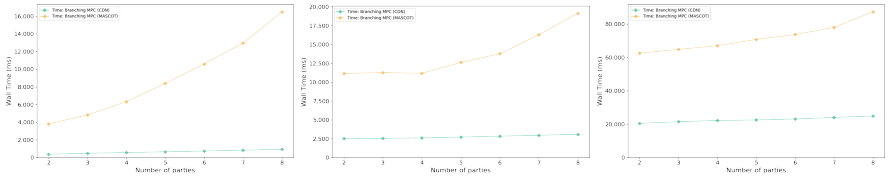
(a) Local (0ms) (b) LAN (10ms) (c) WAN (100ms)

Fig. 6: Running time of Branching MPC with CDN.



(a) Local (0ms) (b) LAN (10ms) (c) WAN (100ms)

Fig. 7: Running time of Branching MPC with Semi-Honest MASCOT.



(a) Local (0ms) (b) LAN (10ms) (c) WAN (100ms)

Fig. 8: Running time of Branching MPC for Different Number of Parties.

8 Acknowledgements

We thank the anonymous reviewers of EUROCRYPT 2022 for their helpful comments. The first, third and forth authors are supported in part by an NSF CNS grant 1814919, NSF CAREER award 1942789 and Johns Hopkins University Catalyst award. The second author is funded by Concordium Blockhain Research Center, Aarhus University, Denmark. The third author is additionally supported by NSF CNS-1653110, NSF CNS-1801479, a Google Security & Privacy Award and DARPA under Agreements No. HR00112020021 and Agreements No. HR001120C0084. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

References

1. Lattigo v2.2.0. Online: `http://github.com/ldsec/lattigo`, July 2021. EPFL-LDS.
2. Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.
3. Gabrielle Beck, Aarushi Goel, Abhishek Jain, and Gabriel Kaptchuk. Order-C secure multiparty computation for highly repetitive circuits. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 663–693. Springer, Heidelberg, October 2021.
4. Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Efficient scalable constant-round MPC via garbled circuits. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 471–498. Springer, Heidelberg, December 2017.
5. 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.
6. Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 410–428. Springer, Heidelberg, August 2013.
7. Elette Boyle, Niv Gilboa, and Yuval Ishai. Breaking the circuit size barrier for secure computation under DDH. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 509–539. Springer, Heidelberg, August 2016.
8. Guilhem Castagnos and Fabien Laguillaumie. Linearly homomorphic encryption from DDH. In Kaisa Nyberg, editor, *CT-RSA 2015*, volume 9048 of *LNCS*, pages 487–505. Springer, Heidelberg, April 2015.
9. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (abstract) (informal contribution). In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, page 462. Springer, Heidelberg, August 1988.
10. Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, abhi shelat, Muthu Venkitasubramaniam, and Ruihan Wang. Diogenes: Lightweight scalable RSA modulus generation with a dishonest majority. Cryptology ePrint Archive, Report 2020/374, 2020. `https://eprint.iacr.org/2020/374`.
11. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 34–64. Springer, Heidelberg, August 2018.
12. Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPDZ2k: Efficient MPC mod 2^k for dishonest majority. Cryptology ePrint Archive, Report 2018/482, 2018. `https://eprint.iacr.org/2018/482`.
13. Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 280–299. Springer, Heidelberg, May 2001.
14. Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 378–394. Springer, Heidelberg, August 2005.

15. Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 119–136. Springer, Heidelberg, February 2001.
16. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.
17. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `https://eprint.iacr.org/2012/144`.
18. Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to mpc with preprocessing using ot. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015*, pages 711–735, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
19. Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.
20. 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.
21. S. Dov Gordon, Daniel Starin, and Arkady Yerukhimovich. The more the merrier: Reducing the cost of large scale MPC. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 694–723. Springer, Heidelberg, October 2021.
22. Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. ATLAS: Efficient and scalable MPC in the honest majority setting. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 244–274, Virtual Event, August 2021. Springer, Heidelberg.
23. Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Unconditional communication-efficient MPC via hall's marriage theorem. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 275–304, Virtual Event, August 2021. Springer, Heidelberg.
24. Vipul Goyal and Yifan Song. Malicious security comes free in honest-majority MPC. Cryptology ePrint Archive, Report 2020/134, 2020. `https://eprint.iacr.org/2020/134`.
25. Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Concretely efficient large-scale MPC with active security (or, TinyKeys for TinyOT). In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 86–117. Springer, Heidelberg, December 2018.
26. David Heath and Vladimir Kolesnikov. Stacked garbling - garbled circuit proportional to longest execution path. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 763–792. Springer, Heidelberg, August 2020.
27. David Heath and Vladimir Kolesnikov. LogStack: Stacked garbling with $O(b \log b)$ computation. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part III*, volume 12698 of *LNCS*, pages 3–32. Springer, Heidelberg, October 2021.
28. David Heath, Vladimir Kolesnikov, and Stanislav Peceny. MOTIF: (almost) free branching in GMW - via vector-scalar multiplication. In Shiho Moriai and Huax-

iong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 3–30. Springer, Heidelberg, December 2020.
29. David Heath, Vladimir Kolesnikov, and Stanislav Peceny. Garbling, stacked and staggered - faster k-out-of-n garbled function evaluation. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part II*, volume 13091 of *Lecture Notes in Computer Science*, pages 245–274. Springer, 2021.
30. David Heath, Vladimir Kolesnikov, and Stanislav Peceny. Masked triples - amortizing multiplication triples across conditionals. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 319–348. Springer, Heidelberg, May 2021.
31. Jonathan Katz and Lior Malka. Constant-round private function evaluation with linear complexity. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 556–571. Springer, Heidelberg, December 2011.
32. Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.
33. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016.
34. Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 158–189. Springer, Heidelberg, April / May 2018.
35. Payman Mohassel and Seyed Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 557–574. Springer, Heidelberg, May 2013.
36. Payman Mohassel, Seyed Saeed Sadeghian, and Nigel P. Smart. Actively secure private function evaluation. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 486–505. Springer, Heidelberg, December 2014.
37. Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 327–346. Springer, Heidelberg, May 1999.
38. Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 554–571. Springer, Heidelberg, August 2008.
39. Ryan Wails, Aaron Johnson, Daniel Starin, Arkady Yerukhimovich, and S. Dov Gordon. Stormy: Statistics in tor by measuring securely. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 615–632. ACM Press, November 2019.
40. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.