Efficient Constant Round Multi-Party Computation Combining BMR and SPDZ

Yehuda Lindell[1], Benny Pinkas[1], Nigel P. Smart[2], and Avishay Yanai[1]

[1] Dept. Computer Science, Bar-Ilan University, Israel,
[2] Dept. Computer Science, University of Bristol, UK

Abstract. Recently, there has been huge progress in the field of concretely efficient secure computation, even while providing security in the presence of *malicious adversaries.* This is especially the case in the two-party setting, where constant-round protocols exist that remain fast even over slow networks. However, in the multi-party setting, all concretely efficient fully-secure protocols, such as SPDZ, require many rounds of communication.

In this paper, we present an MPC protocol that is fully-secure in the presence of malicious adversaries and for any number of corrupted parties. Our construction is based on the constant-round BMR protocol of Beaver et al., and is the first fully-secure version of that protocol that makes black-box usage of the underlying primitives, and is therefore concretely efficient.

Our protocol includes an online phase that is extremely fast and mainly consists of each party locally evaluating a garbled circuit. For the offline phase we present both a generic construction (using any underlying MPC protocol), and a highly efficient instantiation based on the SPDZ protocol. Our estimates show the protocol to be considerably more efficient than previous fully-secure multi-party protocols.

1 Introduction

Background: Protocols for secure multi-party computation (MPC) enable a set of mutually distrustful parties to securely compute a joint functionality of their inputs. Such a protocol must guarantee *privacy* (meaning that only the output is learned), *correctness* (meaning that the output is correctly computed from the inputs), and *independence of inputs* (meaning that each party must choose its input independently of the others). Formally, security is defined by comparing the distribution of the outputs of all parties in a real protocol to an ideal model where an incorruptible trusted party computes the functionality for the parties. The two main types of adversaries that have been considered are *semi-honest adversaries* who follow the protocol specification but try to learn more than allowed by inspecting the transcript, and *malicious adversaries* who can run any arbitrary strategy in an attempt to break the protocol. Secure MPC has been studied since the late 1980s, and powerful feasibility results were proven showing that *any* two-party or multi-party functionality can be securely computed [21, 10], even in the presence of malicious adversaries. When an honest majority (or 2/3 majority) is assumed, then security can even be obtained information theoretically [3, 4, 18]. In this paper, we focus on the problem of security in the presence of malicious adversaries, and a dishonest majority.

Recently, there has been much interest in the problem of concretely efficient secure MPC, where "concretely efficient" refers to protocols that are sufficiently efficient to be implemented in practice (in particular, these protocols should only make black-box usage of cryptographic primitives; they must not, say, use generic ZK proofs that operate on the circuit representation of these primitives). In the last few years there has been tremendous progress on this problem, and there now exist extremely fast protocols that can be used in practice; see [14–16, 13, 8] for just a few examples. In general, there are two approaches that have been followed; the first uses Yao's garbled circuits [21] and the second utilizes interaction for every gate like the GMW protocol [10].

There are extremely efficient variants of Yao's protocol for the two party case that are secure against malicious adversaries (e.g., [14, 15]). These protocols run in a constant number of rounds and therefore remain fast over slow networks. The BMR protocol [1] is a variant of Yao's protocol that runs in a multi-party setting with more than two parties. This protocol works by the parties jointly constructing a garbled circuit (possibly in an offline phase), and then later computing it (possibly in an online phase). However, in the case of malicious adversaries this protocol suffers from two main drawbacks: (1) Security is only guaranteed if at most a *minority* of the parties are corrupt; (2) The protocol uses generic protocols secure against malicious adversaries (say, the GMW protocol) that evaluate the pseudorandom generator used in the BMR protocol. This non black-box construction results in an extremely high overhead.

The TinyOT and SPDZ protocols [16, 8] follow the GMW paradigm, and have offline and online phases. Both of these protocols overcome the issues of the BMR protocol in that they are secure against any number of corrupt parties, make only black-box usage of cryptographic primitives, and have very fast online phases that require only very simple (information theoretic) operations. (A black-box constant-round MPC construction appears in [11]; however, it is not "concretely efficient".) In the case of multi-party computation with more than two parties, these protocols are currently the *only practical* approach known. However, since they follow the GMW paradigm, their online phase requires a communication round for every multiplication gate. This results in a large amount of interaction and high latency, especially over slow networks. To sum up, there is no known concretely efficient constant-round protocol for the multi-party case (with the exception of [5] that considers the specific three-party case only). Our work introduces the first protocol with these properties.

Our contribution: In this paper, we provide the first *concretely efficient constant-round* protocol for the general *multi-party* case, with security in the presence of malicious adversaries. The basic idea behind the construction is to use an efficient non-constant round protocol – with security for malicious adversaries – to compute the gate tables of the BMR garbled circuit (and since the computation of these tables is of constant depth, this step is constant round). A crucial observation, resulting in a great performance improvement, shows that in the offline stage it is *not required to verify the correctness* of the computations of the different tables. Rather, validation of the correctness is an immediate by product of the online computation phase, and therefore does not add any overhead to the computation. Although our basic generic protocol can be instantiated with any non-constant round MPC protocol, we provide an optimized version that utilizes specific features of the SPDZ protocol [8].

In our general construction, the new constant-round MPC protocol consists of two phases. In the first (offline) phase, the parties securely compute *random shares* of the BMR garbled circuit. If this is done naively, then the result is highly inefficient since part of the computation involves computing a pseudorandom generator or pseudorandom function multiple times for every gate. By modifying the original BMR garbled circuit, we show that it is possible to actually compute the circuit very efficiently. Specifically, each party locally computes the pseudorandom function as needed for every gate (in our construction we use a pseudorandom function rather than a pseudorandom generator), and uses the results as input to the secure computation. Our proof of security shows that if a party cheats and inputs incorrect values then no harm is done, since it can only cause the honest parties to abort (which is anyway possible when there is no honest majority). Next, in the online phase, all that the parties need to do is reconstruct the single garbled circuit, exchange garbled values on the input wires and locally compute the garbled circuit. The online phase is therefore very fast.

In our concrete instantiation of the protocol using SPDZ [8], there are actually three separate phases, with each being faster than the previous. The first two phases can be run offline, and the last phase is run online after the inputs become known.

- The first (slow) phase depends only on an upper bound on the number of wires and the number of gates in the function to be evaluated. This phase uses Somewhat Homomorphic Encryption (SHE) and is equivalent to the offline phase of the SPDZ protocol.
- The second phase depends on the function to be evaluated but not the function inputs; in our proposed instantiation this mainly involves information theoretic primitives and is equivalent to the online phase of the SPDZ protocol.
- In the third phase the parties provide their input and evaluate the function; this phase just involves exchanging shares of the circuit and garbled values on the input wire and locally computing the BMR garbled circuit.

We stress that our protocol is constant round *in all phases* since the depth of the circuit required to compute the BMR garbled circuit is constant. In addition, the computational cost of preparing the BMR garbled circuit is not much more than the cost of using SPDZ itself to compute the functionality directly. However, the key advantage that we gain is that our online time is extraordinarily fast, requiring only two rounds and local computation of a single garbled circuit. *This is faster than all other existing circuit-based multi-party protocols.*

Finite field optimization of BMR: In order to efficiently compute the BMR garbled circuit, we define the garbling and evaluation operations over a finite field. A similar technique of using finite fields in the BMR protocol was introduced in [2] in the case of semi-honest security against an honest majority. In contrast to [2], our utilization of finite fields is carried out via *vectors* of field elements, and uses the underlying arithmetic of the field as opposed to using very large finite fields to simulate integer arithmetic. This makes our modification in this respect more efficient.

2 The General Protocol

2.1 Modified BMR Garbling

In order to facilitate fast secure computation of the garbled circuit in the offline phase, we make some changes to the original BMR garbling described in Appendix A. First, instead of using XOR of bit strings, and hence a binary circuit to instantiate the garbled gate, we use additions of elements in a finite field, and hence an arithmetic circuit. This idea was used by [2] in the FairplayMP system, which used the BGW protocol [3] in order to compute the BMR circuit. Note that FairplayMP achieved semi-honest security with an honest majority, whereas our aim is *malicious security* for *any number of corrupted parties*.

Second, we observe that the external values[3] do not need to be explicitly encoded, since each party can learn them by looking at its own "part" of the garbled value. In the original BMR garbling, each superseed contains n seeds provided by the parties. Thus, if a party's zero-seed is in the decrypted superseed then it knows that the external value (denoted by Λ) is zero, and otherwise it knows that it is one.

Naively, it seems that independently computing each gate securely in the offline phase is insufficient, since the corrupted parties might use inconsistent inputs for the computations of different gates. For example, if the output wire of gate g is an input to gate g', the input provided for the computation of the table of g might not agree with the inputs used for the computation of the table of g'. It therefore seems that the offline computation must verify the consistency of the computations of different gates. This type of verification would greatly increase the cost since the evaluation of the pseudorandom functions (or pseudorandom generator in the original BMR) used in computing the tables needs to be be checked inside the secure computation. This means that the pseudorandom function is not treated as a black box, and the circuit for the offline phase would be huge (as it would include multiple copies of a subcircuit for computing pseudorandom function computations for every wire). Instead, we prove that this type of corrupt behavior can only result in an abort in the online phase, which would not affect the security of the protocol. This observation enables us to compute each gate independently and model the pseudorandom

[3] The external values (as denoted in [2]) are the *signals* (as denoted in [1]) observable by the parties when evaluating the circuit in the online phase.

function used in the computation as a black box, thus simplifying the protocol and optimizing its performance.

We also encrypt garbled values as *vectors*; this enables us to use a finite field that can encode $\{0,1\}^\kappa$ (for each vector coordinate), rather than a much larger finite field that can encode all of $\{0,1\}^{n\cdot\kappa}$. Due to this, the parties choose *keys* (for a pseudorandom function) rather than *seeds* for a pseudorandom generator. The keys that P_i chooses for wire w are denoted $k^i_{w,0}$ and $k^i_{w,1}$, which will be elements in a finite field $\mathbb{F}_p$ such that $2^\kappa < p < 2^{\kappa+1}$. In fact we pick p to be the smallest prime number larger than 2^κ, and set $p = 2^\kappa + \alpha$, where (by the prime number theorem) we expect $\alpha \approx \kappa$. We shall denote the pseudorandom function by $F_k(x)$, where the key and output will be interpreted as elements of $\mathbb{F}_p$ in much of our MPC protocol. In practice the function $F_k(x)$ we suggest will be implemented using CBC-MAC using a block cipher enc with key and block size κ bits, as $F_k(x) = \mathsf{CBC\text{-}MAC}_{\mathsf{enc}}(k \pmod{2^\kappa}, x)$. Note that the inputs x to our pseudorandom function will all be of the same length and so using naive CBC-MAC will be secure.

We interpret the κ-bit output of $F_k(x)$ as an element in $\mathbb{F}_p$ where $p = 2^\kappa + \alpha$. Note that a mapping which sends an element $k \in \mathbb{F}_p$ to a κ-bit block cipher key by computing $k \pmod{2^\kappa}$ induces a distribution on the key space of the block cipher which has statistical distance from uniform of

$$\frac{1}{2}\left((2^\kappa - \alpha)\cdot\left(\frac{1}{2^\kappa} - \frac{1}{p}\right) + \alpha\cdot\left(\frac{2}{p} - \frac{1}{2^\kappa}\right)\right) \approx \frac{\alpha}{p} \approx \frac{\kappa}{2^\kappa}.$$

The output of the function $F_k(x)$ will also induce a distribution which is close to uniform on $\mathbb{F}_p$. In particular the statistical distance of the output in $\mathbb{F}_p$, for a block cipher with block size κ, from uniform is given by

$$\frac{1}{2}\left(2^\kappa\cdot\left(\frac{1}{2^\kappa} - \frac{1}{p}\right) + \alpha\cdot\left(\frac{1}{p} - 0\right)\right) = \frac{\alpha}{p} \approx \frac{\kappa}{2^\kappa}$$

(note that $1 - \frac{2^\kappa}{p} = \frac{\alpha}{p}$). In practice we set $\kappa = 128$, and use the AES cipher as the block cipher enc. The statistical difference is therefore negligible.

The goal of this paper is to present a protocol Π_{SFE} which implements the Secure Function Evaluation (SFE) functionality of Functionality 1 in a constant number of rounds in the case of a malicious dishonest majority. Our constant round protocol Π_{SFE} implementing $\mathcal{F}_{\mathsf{SFE}}$ is built in the $\mathcal{F}_{\mathsf{MPC}}$-hybrid model, i.e. utilizing a sub-protocol Π_{MPC} which implements the functionality $\mathcal{F}_{\mathsf{MPC}}$ given in Functionality 2. The generic MPC functionality $\mathcal{F}_{\mathsf{MPC}}$ is *reactive*. We require a *reactive* MPC functionality because our protocol Π_{SFE} will make repeated sequences of calls to $\mathcal{F}_{\mathsf{MPC}}$ involving both output and computation commands. In terms of round complexity, all that we require of the sub-protocol Π_{MPC} is that each of the commands which it implements can be implemented in constant rounds. Given this requirement our larger protocol Π_{SFE} will be constant round.

Functionality 1 (The SFE Functionality: $\mathcal{F}_{\mathsf{SFE}}$)

The functionality is parameterized by a function $f(x_1, \ldots, x_n)$ which is input as a binary circuit C_f. The protocol consists of 3 externally exposed commands **Initialize**, **InputData**, and **Output** and one internal subroutine **Wait**.

Initialize: On input $(init, C_f)$ from all parties, the functionality activates and stores C_f.

Wait: This waits on the adversary to return a *GO*/*NO-GO* decision. If the adversary returns *NO-GO* then the functionality aborts.

InputData: On input $(input, P_i, varid, x_i)$ from P_i and $(input, P_i, varid, ?)$ from all other parties, with *varid* a fresh identifier, the functionality stores $(varid, x_i)$. The functionality then calls **Wait**.

Output: On input $(output)$ from all honest parties the functionality computes $y = f(x_1, \ldots, x_n)$ and outputs y to the adversary. The functionality then calls **Wait**. Only if **Wait** does not abort it outputs y to all parties.

Functionality 2 (The Generic Reactive MPC Functionality: $\mathcal{F}_{\mathsf{MPC}}$)

The functionality consists of five externally exposed commands **Initialize**, **InputData**, **Add**, **Multiply**, and **Output**, and one internal subroutine **Wait**.

Initialize: On input $(init, p)$ from all parties, the functionality activates and stores p. All additions and multiplications below will be mod p.

Wait: This waits on the adversary to return a *GO*/*NO-GO* decision. If the adversary returns *NO-GO* then the functionality aborts.

InputData: On input $(input, P_i, varid, x)$ from P_i and $(input, P_i, varid, ?)$ from all other parties, with *varid* a fresh identifier, the functionality stores $(varid, x)$. The functionality then calls **Wait**.

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 $(varid_3, x + y \bmod p)$. The functionality then calls **Wait**.

Multiply: On input $(multiply, varid_1, varid_2, varid_3)$ 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 $(varid_3, x \cdot y \bmod p)$. The functionality then calls **Wait**.

Output: On input $(output, varid, i)$ from all honest parties (if *varid* is present in memory), the functionality retrieves $(varid, x)$ and outputs either $(varid, x)$ in the case of $i \neq 0$ or $(varid)$ if $i = 0$ to the adversary. The functionality then calls **Wait**, and only if **Wait** does not abort then it outputs x to all parties if $i = 0$, or it outputs x only to party i if $i \neq 0$.

In what follows we use the notation $[varid]$ to represent the result stored in the variable *varid* by the $\mathcal{F}_{\mathsf{MPC}}$ or $\mathcal{F}_{\mathsf{SFE}}$ functionality. In particular we use the arithmetic shorthands $[z] = [x] + [y]$ and $[z] = [x] \cdot [y]$ to represent the result of calling the **Add** and **Multiply** commands on the $\mathcal{F}_{\mathsf{MPC}}$ functionality.

2.2 The Offline Functionality: preprocessing-I and preprocessing-II

Our protocol, $\mathit{\Pi}_{\mathsf{SFE}}$, is comprised of an offline-phase and an online-phase, where the offline-phase, which implements the functionality $\mathcal{F}_{\mathsf{offline}}$, is divided into two subphases: preprocessing-I and preprocessing-II. To aid exposition we first present the functionality $\mathcal{F}_{\mathsf{offline}}$ in Functionality 3. In the next section, we present an efficient methodology to implement $\mathcal{F}_{\mathsf{offline}}$ which uses the SPDZ protocol as the underlying MPC protocol for securely computing functionality $\mathcal{F}_{\mathsf{MPC}}$; while in Appendix B we present a generic implementation of $\mathcal{F}_{\mathsf{offline}}$ based on any underlying protocol $\mathit{\Pi}_{\mathsf{MPC}}$ implementing $\mathcal{F}_{\mathsf{MPC}}$.

In describing functionality $\mathcal{F}_{\mathsf{offline}}$ we distinguish between *attached* wires and *common* wires: the attached wires are the circuit-input-wires that are directly connected to the parties (i.e., these are inputs wires to the circuit). Thus, if every party has ℓ inputs to the functionality f then there are $n \cdot \ell$ attached wires. The rest of the wires are considered as *common* wires, i.e. they are directly connected to *none* of the parties.

Our preprocessing-I takes as input an upper bound W on the number of wires in the circuit, and an upper bound G on the number of gates in the circuit. The upper bound G is not strictly needed, but will be needed in any efficient instantiation based on the SPDZ protocol. In contrast preprocessing-II requires knowledge of the precise function f being computed, which we assume is encoded as a binary circuit C_f.

In order to optimize the performance of the preprocessing-II phase, the secure computation does not evaluate the pseudorandom function $F()$, but rather has the parties compute $F()$ and provide the results as an input to the protocol. Observe that corrupted parties may provide *incorrect* input values $F_{k^i_{x,j}}()$ and thus the resulting garbled circuit may not actually be a valid BMR garbled circuit. Nevertheless, we show that such behavior can only result in an abort. This is due to the fact that if a value is incorrect and honest parties see that their key (coordinate) is not present in the resulting vector then they will abort. In contrast, if their seed is present then they proceed and the incorrect value had no effect. Since the keys are secret, the adversary cannot give an incorrect value that will result in a correct *different* key, except with negligible probability. This is important since otherwise correctness would be harmed. Likewise, a corrupted party cannot influence the masking values λ, and thus they are consistent throughout (when a given wire is input into multiple gates), ensuring correctness.

2.3 Securely Computing $\mathcal{F}_{\mathsf{SFE}}$ in the $\mathcal{F}_{\mathsf{offline}}$-Hybrid Model

We now define our protocol $\mathit{\Pi}_{\mathsf{SFE}}$ for securely computing $\mathcal{F}_{\mathsf{SFE}}$ (using the BMR garbled circuit) in the $\mathcal{F}_{\mathsf{offline}}$-hybrid model, see Protocol 1.

2.4 Implementing $\mathcal{F}_{\mathsf{offline}}$ in the $\mathcal{F}_{\mathsf{MPC}}$-Hybrid Model

At first sight, it may seem that in order to construct an entire garbled circuit (i.e. the output of $\mathcal{F}_{\mathsf{offline}}$), an ideal functionality that computes each garbled gate can

Functionality 3 (The Offline Functionality – $\mathcal{F}_{\mathsf{offline}}$)

This functionality runs the same **Initialize**, **Wait**, **InputData** and **Output** commands as $\mathcal{F}_{\mathsf{MPC}}$ (Functionality 2). In addition, the functionality has two additional commands **preprocessing-I** and **preprocessing-II**, as follows.

preprocessing-I: On input $(\mathsf{preprocessing\text{-}I}, W, G)$, for all wires $w \in [1, \ldots, W]$:

- The functionality chooses and stores a random masking value $[\lambda_w]$ where $\lambda_w \in \{0, 1\}$.
- For $1 \leq i \leq n$ and $\beta \in \{0, 1\}$,
  - The functionality stores a key of user i for wire w and value β, $[k^i_{w,\beta}]$ where $k^i_{w,\beta} \in \mathbb{F}_p$
  - The functionality outputs $[k^i_{w,\beta}]$ to party i by running **Output** as in functionality $\mathcal{F}_{\mathsf{MPC}}$.

preprocessing-II: On input of $(\mathsf{preprocessing\text{-}II}, C_f)$ for a circuit C_f with at most W wires and G gates.

- For all wires w which are attached to party P_i the functionality opens $[\lambda_w]$ to party P_i by running **Output** as in functionality $\mathcal{F}_{\mathsf{MPC}}$.
- For all output wires w the functionality opens $[\lambda_w]$ to all parties by running **Output** as in functionality $\mathcal{F}_{\mathsf{MPC}}$.
- For every gate g with input wires $1 \leq a, b \leq W$ and output wire $1 \leq c \leq W$.
  - Party P_i provides the following values for $x \in \{a, b\}$ by running **InputData** as in functionality $\mathcal{F}_{\mathsf{MPC}}$:

$$F_{k^i_{x,0}}(0\|1\|g), \ldots, F_{k^i_{x,0}}(0\|n\|g) \qquad F_{k^i_{x,0}}(1\|1\|g), \ldots, F_{k^i_{x,0}}(1\|n\|g)$$
$$F_{k^i_{x,1}}(0\|1\|g), \ldots, F_{k^i_{x,1}}(0\|n\|g) \qquad F_{k^i_{x,1}}(1\|1\|g), \ldots, F_{k^i_{x,1}}(1\|n\|g)$$

  - Define the selector variables

$$\chi_1 = \begin{cases} 0 & \text{if } f_g(\lambda_a, \lambda_b) = \lambda_c \\ 1 & \textit{otherwise} \end{cases} \qquad \chi_2 = \begin{cases} 0 & \text{if } f_g(\lambda_a, \overline{\lambda_b}) = \lambda_c \\ 1 & \textit{otherwise} \end{cases}$$

$$\chi_3 = \begin{cases} 0 & \text{if } f_g(\overline{\lambda_a}, \lambda_b) = \lambda_c \\ 1 & \textit{otherwise} \end{cases} \qquad \chi_4 = \begin{cases} 0 & \text{if } f_g(\overline{\lambda_a}, \overline{\lambda_b}) = \lambda_c \\ 1 & \textit{otherwise} \end{cases}$$

  - Set $\mathbf{A}_g = (A^1_g, \ldots, A^n_g)$, $\mathbf{B}_g = (B^1_g, \ldots, B^n_g)$, $\mathbf{C}_g = (C^1_g, \ldots, C^n_g)$, and $\mathbf{D}_g = (D^1_g, \ldots, D^n_g)$ where for $1 \leq j \leq n$:

$$A^j_g = \left(\sum_{i=1}^{n} F_{k^i_{a,0}}(0\|j\|g) + F_{k^i_{b,0}}(0\|j\|g) \right) + k^j_{c,\chi_1}$$

$$B^j_g = \left(\sum_{i=1}^{n} F_{k^i_{a,0}}(1\|j\|g) + F_{k^i_{b,1}}(0\|j\|g) \right) + k^j_{c,\chi_2}$$

$$C^j_g = \left(\sum_{i=1}^{n} F_{k^i_{a,1}}(0\|j\|g) + F_{k^i_{b,0}}(1\|j\|g) \right) + k^j_{c,\chi_3}$$

$$D^j_g = \left(\sum_{i=1}^{n} F_{k^i_{a,1}}(1\|j\|g) + F_{k^i_{b,1}}(1\|j\|g) \right) + k^j_{c,\chi_4}$$

  - The functionality stores the values $[\mathbf{A}_g], [\mathbf{B}_g], [\mathbf{C}_g], [\mathbf{D}_g]$.

Protocol 1 (Π_{SFE}: Securely Computing $\mathcal{F}_{\mathsf{SFE}}$ in the $\mathcal{F}_{\mathsf{offline}}$-Hybrid Model)

On input of a circuit C_f representing the function f which consists of at most W wires and at most G gates the parties execute the following commands.

Pre-Processing: This procedure is performed as follows

1. Call **Initialize** on $\mathcal{F}_{\mathsf{offline}}$ with the smallest prime p in $\{2^\kappa, \ldots, 2^{\kappa+1}\}$.
2. Call **Preprocessing-I** on $\mathcal{F}_{\mathsf{offline}}$ with input W and G.
3. Call **Preprocessing-II** on $\mathcal{F}_{\mathsf{offline}}$ with input C_f.

Online Computation: This procedure is performed as follows

1. For all input wires w for party P_i the party takes his input bit ρ_w and computes $\Lambda_w = \rho_w \oplus \lambda_w$, where λ_w was obtained in the preprocessing stage. The value Λ_w is broadcast to all parties.
2. Party i calls **Output** on $\mathcal{F}_{\mathsf{offline}}$ to open $[k^i_{w,\Lambda_w}]$ for all his input wires w, we denote the resulting value by k^i_w.
3. The parties call **Output** on $\mathcal{F}_{\mathsf{offline}}$ to open $[\mathbf{A}_g]$, $[\mathbf{B}_g]$, $[\mathbf{C}_g]$ and $[\mathbf{D}_g]$ for every gate g.
4. Passing through the circuit topologically, the parties can now locally compute the following operations for each gate g
   - Let the gates input wires be labeled a and b, and the output wire be labeled c.
   - For $j = 1, \ldots, n$ compute k^j_c according to the following cases:
     - *Case 1* – $(\Lambda_a, \Lambda_b) = (0, 0)$*:* compute
       $$k^j_c = A^j_g - \left(\sum_{i=1}^{n} F_{k^i_a}(0\|j\|g) + F_{k^i_b}(0\|j\|g)\right).$$
     - *Case 2* – $(\Lambda_a, \Lambda_b) = (0, 1)$*:* compute
       $$k^j_c = B^j_g - \left(\sum_{i=1}^{n} F_{k^i_a}(1\|j\|g) + F_{k^i_b}(0\|j\|g)\right).$$
     - *Case 3* – $(\Lambda_a, \Lambda_b) = (1, 0)$*:* compute
       $$k^j_c = C^j_g - \left(\sum_{i=1}^{n} F_{k^i_a}(0\|j\|g) + F_{k^i_b}(1\|j\|g)\right).$$
     - *Case 4* – $(\Lambda_a, \Lambda_b) = (1, 1)$*:* compute
       $$k^j_c = D^j_g - \left(\sum_{i=1}^{n} F_{k^i_a}(1\|j\|g) + F_{k^i_b}(1\|j\|g)\right).$$
   - If $k^i_c \notin \{k^i_{c,0}, k^i_{c,1}\}$, then P_i outputs abort. Otherwise, it proceeds. If P_i aborts it notifies all other parties with that information. If P_i is notified that another party has aborted it aborts as well.
   - If $k^i_c = k^i_{c,0}$ then P_i sets $\Lambda_c = 0$; if $k^i_c = k^i_{c,1}$ then P_i sets $\Lambda_c = 1$.
   - The output of the gate is defined to be $(k^1_c, \ldots, k^n_c)$ and Λ_c.
5. Assuming party P_i does not abort it will obtain Λ_w for every circuit-output wire w. The party can then recover the actual output value from $\rho_w = \Lambda_w \oplus \lambda_w$, where λ_w was obtained in the preprocessing stage.

be used separately for each gate of the circuit (that is, for each gate the parties provide their PRF results on the keys and shares of the masking values associated with that gate's wires). This is sufficient when considering semi-honest adversaries. However, in the setting of malicious adversaries, this can be problematic since parties may input inconsistent values. For example, the masking values λ_w that are common to a number of gates (which happens when any wire enters more than one gate) need to be identical in all of these gates. In addition, the pseudorandom function values may not be correctly computed from the pseudorandom function keys that are input. In order to make the computation of the garbled circuit efficient, we will not check that the pseudorandom function values are correct. However, it is necessary to ensure that the λ_w values are correct, and that they (and likewise the keys) are consistent between gates (e.g., as in the case where the same wire is input to multiple gates). We achieve this by computing the entire circuit at once, via a single functionality.

The cost of this computation is actually almost the same as separately computing each gate. The single functionality receives from party P_i the values $k^i_{w,0}, k^i_{w,1}$ and the output of the pseudorandom function applied to the keys *only once*, regardless of the number of gates to which w is input. Thereby consistency is immediate throughout, and this potential attack is prevented. Moreover, the λ_w values are generated once and used consistently by the circuit, making it easy to ensure that the λ values are correct.

Another issue that arises is that the single garbled gate functionality expects to receive a single masking value for each wire. However, since this value is secret, it must be generated from shares that are input by the parties. In Appendix B we describe the full protocol for securely computing $\mathcal{F}_{\mathsf{offline}}$ in the $\mathcal{F}_{\mathsf{MPC}}$-hybrid model (i.e., using *any* protocol that securely computes the $\mathcal{F}_{\mathsf{MPC}}$ ideal functionality). In short, the parties input shares of λ_w to the functionality, the single masking value is computed from these shares, and then input to all the necessary gates.

In the semi-honest case, the parties could contribute a share which is random in $\{0,1\}$ (interpreted as an element in $\mathbb{F}_p$) and then compute the product of all the shares (using the underlying MPC) to obtain a random masking value in $\{0,1\}$. This is however not the case in the malicious case since parties might provide a share that is not from $\{0,1\}$ and thus the resulting masking value wouldn't likewise be from $\{0,1\}$

This issue is solved in the following way. The computation is performed by having the parties input random masking values $\lambda^i_w \in \{1,-1\}$, instead of bits. This enables the computation of a value μ_w to be the *product* of $\lambda^1_w,\ldots,\lambda^n_w$ and to be random in $\{-1,1\}$ as long as one of them is random. The product is then mapped to $\{0,1\}$ in $\mathbb{F}_p$ by computing $\lambda_w = \frac{\mu_w+1}{2}$.

In order to prevent corrupted parties from inputting λ^i_w values that are not in $\{-1,+1\}$, the protocol for computing the circuit outputs $(\prod_{i=1}^n \lambda^i_w)^2 - 1$, for every wire w (where λ^i_w is the share contributed from party i for wire w), and the parties can simply check whether it is equal to zero or not. Thus, if any party cheats by causing some $\lambda_w \notin \{-1,+1\}$, then this will be discovered since the circuit outputs a non-zero value for $(\prod_{i=1}^n \lambda^i_w)^2 - 1$, and so the parties detect this and can abort. Since this occurs before any inputs are used, nothing is revealed by this. Furthermore, if $\prod_{i=1}^{n} \lambda_w^i \in \{-1, +1\}$, then the additional value output reveals nothing about λ_w itself.

In the next section we shall remove *all* of the complications by basing our implementation for $\mathcal{F}_{\mathsf{MPC}}$ upon the specific SPDZ protocol. The reason why the SPDZ implementation is simpler – and more efficient – is that SPDZ provides generation of such shared values effectively for free.

3 The SPDZ Based Instantiation

Functionality 4 (The SPDZ Functionality: $\mathcal{F}_{\mathsf{SPDZ}}$)

The functionality consists of seven externally exposed commands **Initialize**, **InputData**, **RandomBit**, **Random**, **Add**, **Multiply**, and **Output** and one internal subroutine **Wait**.

Initialize: On input $(init, p, M, B, R, I)$ from all parties, the functionality activates and stores p. Pre-processing is performed to generate data needed to respond to a maximum of M **Multiply**, B **RandomBit**, R **Random** commands, and I **InputData** commands per party.

Wait: This waits on the adversary to return a *GO*/*NO-GO* decision. If the adversary returns *NO-GO* then the functionality aborts.

InputData: On input $(input, P_i, varid, x)$ from P_i and $(input, P_i, varid, ?)$ from all other parties, with *varid* a fresh identifier, the functionality stores $(varid, x)$. The functionality then calls **Wait**.

RandomBit: On command $(randombit, varid)$ from all parties, with *varid* a fresh identifier, the functionality selects a random value $r \in \{0, 1\}$ and stores $(varid, r)$. The functionality then calls **Wait**.

Random: On command $(random, varid)$ from all parties, with *varid* a fresh identifier, the functionality selects a random value $r \in \mathbb{F}_p$ and stores $(varid, r)$. The functionality then calls **Wait**.

Add: On command $(add, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$, stores $(varid_3, x + y)$ and then calls **Wait**.

Multiply: On input $(multiply, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$, stores $(varid_3, x \cdot y)$ and then calls **Wait**.

Output: On input $(output, varid, i)$ from all honest parties (if *varid* is present in memory), the functionality retrieves $(varid, x)$ and outputs either $(varid, x)$ in the case of $i \neq 0$ or $(varid)$ if $i = 0$ to the adversary. The functionality then calls **Wait**, and only if **Wait** does not abort then it outputs x to all parties if $i = 0$, or it outputs x only to party i if $i \neq 0$.

3.1 Utilizing the SPDZ Protocol

As discussed in Section 2.1, in the offline-phase we use an underlying secure computation protocol, which, given a binary circuit and the matching inputs to its input wires, securely and distributively computes that binary circuit. In this section we simplify and optimize the implementation of the protocol Π_{offline} which implements the functionality $\mathcal{F}_{\mathsf{offline}}$ by utilizing the specific SPDZ MPC protocol as the underlying implementation of $\mathcal{F}_{\mathsf{MPC}}$. These optimizations are possible because the SPDZ MPC protocol provides a richer interface to the protocol designer than the naive generic MPC interface given in functionality $\mathcal{F}_{\mathsf{MPC}}$. In particular, it provides the capability of directly generating shared random bits and strings. These are used for generating the masking values and pseudorandom function keys. Note that one of the most expensive steps in FairplayMP [2] was coin tossing to generate the masking values; by utilizing the specific properties of SPDZ this is achieved essentially for free.

In Section 3.2 we describe explicit operations that are to be carried out on the inputs in order to achieve the desired output; the circuit's complexity analysis appears in Section 3.3 and the expected results from an implementation of the circuit using the SPDZ protocol are in Section 3.4.

Throughout, we utilize $\mathcal{F}_{\mathsf{SPDZ}}$ (Functionality 4), which represents an idealized representation of the SPDZ protocol, akin to the functionality $\mathcal{F}_{\mathsf{MPC}}$ from Section 2.1. Note that in the real protocol, $\mathcal{F}_{\mathsf{SPDZ}}$ is implemented itself by an offline phase (essentially corresponding to our preprocessing-I) and an online phase (corresponding to our preprocessing-II). We fold the SPDZ offline phase into the **Initialize** command of $\mathcal{F}_{\mathsf{SPDZ}}$. In the SPDZ offline phase we need to know the maximum number of multiplications, random values and random bits required in the online phase. In that phase the random shared bits and values are produced, as well as the "Beaver Triples" for use in the multiplication gates performed in the SPDZ online phase. In particular the consuming of shared random bits and values results in no cost during the SPDZ online phase, with all consumption costs being performed in the SPDZ offline phase. The protocol, which utilizes Somewhat Homomorphic Encryption to produce the shared random values/bits and the Beaver multiplication triples, is given in [7].

As before, we use the notation $[varid]$ to represent the result stored in the variable *varid* by the functionality. In particular we use the arithmetic shorthands $[z] = [x] + [y]$ and $[z] = [x] \cdot [y]$ to represent the result of calling the **Add** and **Multiply** commands on the functionality $\mathcal{F}_{\mathsf{SPDZ}}$.

3.2 The Π_{offline} SPDZ based Protocol

As remarked earlier $\mathcal{F}_{\mathsf{offline}}$ can be securely computed using *any* secure multiparty protocol. This is advantageous since it means that future efficiency improvements to concretely secure multi-party computation (with dishonest majority) will automatically make our protocol faster. However, currently the best option is SPDZ. Specifically, it utilizes the fact that SPDZ can very efficiently generate coin tosses. This means that it is not necessary for the parties to input the λ_w^i values, to multiply them together to obtain λ_w and to output the check values $(\lambda_w)^2 - 1$. Thus, this yields a significant efficiency improvement. We now describe the protocol which implements $\mathcal{F}_{\mathsf{offline}}$ in the $\mathcal{F}_{\mathsf{SPDZ}}$-hybrid model

preprocessing-I:

1. **Initialize the MPC Engine:** Call **Initialize** on the functionality $\mathcal{F}_{\mathsf{SPDZ}}$ with input p, a prime with $p > 2^k$ and with parameters
$$M = 13 \cdot G, \quad B = W, \quad R = 2 \cdot W \cdot n, \quad I = 2 \cdot G \cdot n + W,$$
where G is the number of gates, n is the number of parties and W is the number of input wires per party. In practice the term W in the calculation of I needs only be an upper bound on the total number of input wires per party in the circuit which will eventually be evaluated.
2. **Generate wire masks:** For every circuit wire w we need to generate a sharing of the (secret) masking-values λ_w. Thus for *all* wires w the parties execute the command **RandomBit** on the functionality $\mathcal{F}_{\mathsf{SPDZ}}$, the output is denoted by $[\lambda_w]$. The functionality $\mathcal{F}_{\mathsf{SPDZ}}$ guarantees that $\lambda_w \in \{0, 1\}$.
3. **Generate keys:** For every wire w, each party $i \in [1, \ldots, n]$ and for $j \in \{0, 1\}$, the parties call **Random** on the functionality $\mathcal{F}_{\mathsf{SPDZ}}$ to obtain output $[k_{w,j}^i]$. The parties then call **Output** to open $[k_{w,j}^i]$ to party i for all j and w. The vector of shares $[k_{w,j}^i]_{i=1}^n$ we shall denote by $[\mathbf{k}_{w,j}]$.

preprocessing-II: (This protocol implements the computation gate table as it is detailed in the BMR protocol. The correctness of this construction is explained at the end of Appendix A.)

1. **Output input wire values:** For all wires w which are attached to party P_i we execute the command **Output** on the functionality $\mathcal{F}_{\mathsf{SPDZ}}$ to open $[\lambda_w]$ to party i.
2. **Output masks for circuit-output-wires:** In order to reveal the real values of the circuit-output-wires it is required to reveal their masking values. That is, for every circuit-output-wire w, the parties execute the command **Output** on the functionality $\mathcal{F}_{\mathsf{SPDZ}}$ for the stored value $[\lambda_w]$.
3. **Calculate garbled gates:** This step is operated for each gate g in the circuit in parallel. Specifically, let g be a gate whose input wires are a, b and output wire is c. Do as follows:
    (a) **Calculate output indicators:** This step calculates four indicators $[x_a]$, $[x_b]$, $[x_c]$, $[x_d]$ whose values will be in $\{0, 1\}$. Each one of the garbled labels $\mathbf{A}_g, \mathbf{B}_g, \mathbf{C}_g, \mathbf{D}_g$ is a vector of n elements that hide either the vector $\mathbf{k}_{c,0} = k_{c,0}^1, \ldots, k_{c,0}^n$ or $\mathbf{k}_{c,1} = k_{c,1}^1, \ldots, k_{c,1}^n$; which one it hides depends on these indicators, i.e if $x_a = 0$ then $\mathbf{A}_g$ hides $\mathbf{k}_{c,0}$ and if $x_a = 1$ then $\mathbf{A}_g$ hides $\mathbf{k}_{c,1}$. Similarly, $\mathbf{B}_g$ depends on x_b, $\mathbf{C}_g$ depends on x_c and $\mathbf{D}_c$ depends on x_d. Each indicator is determined by some function on $[\lambda_a]$, $[\lambda_b]$,$[\lambda_c]$ and the truth table of the gate f_g. Every indicator is calculated slightly different, as follows (concrete examples are given after

the preprocessing specification):

$$\begin{aligned}
[x_a] &= \left(f_g([\lambda_a],[\lambda_b]) \overset{?}{\neq} [\lambda_c]\right) &= (f_g([\lambda_a],[\lambda_b]) - [\lambda_c])^2\\
[x_b] &= \left(f_g([\lambda_a],\overline{[\lambda_b]}) \overset{?}{\neq} [\lambda_c]\right) &= (f_g([\lambda_a],(1-[\lambda_b])) - [\lambda_c])^2\\
[x_c] &= \left(f_g(\overline{[\lambda_a]},[\lambda_b]) \overset{?}{\neq} [\lambda_c]\right) &= (f_g((1-[\lambda_a]),[\lambda_b]) - [\lambda_c])^2\\
[x_d] &= \left(f_g(\overline{[\lambda_a]},\overline{[\lambda_b]}) \overset{?}{\neq} [\lambda_c]\right) &= (f_g((1-[\lambda_a]),(1-[\lambda_b])) - [\lambda_c])^2
\end{aligned}$$

where the binary operator $\overset{?}{\neq}$ is defined as $[a] \overset{?}{\neq} [b]$ equals $[0]$ if $a = b$, and equals $[1]$ if $a \neq b$. For the XOR function on a and b, for example, the operator can be evaluated by computing $[a] + [b] - 2 \cdot [a] \cdot [b]$. Thus, these can be computed using **Add** and **Multiply**.

(b) **Assign the correct vector:** As described above, we use the calculated indicators to choose for every garbled label either $\mathbf{k}_{c,0}$ or $\mathbf{k}_{c,1}$. Calculate:

$$\begin{aligned}
[\mathbf{v}_{c,x_a}] &= (1-[x_a]) \cdot [\mathbf{k}_{c,0}] + [x_a] \cdot [\mathbf{k}_{c,1}]\\
[\mathbf{v}_{c,x_b}] &= (1-[x_b]) \cdot [\mathbf{k}_{c,0}] + [x_a] \cdot [\mathbf{k}_{c,1}]\\
[\mathbf{v}_{c,x_c}] &= (1-[x_c]) \cdot [\mathbf{k}_{c,0}] + [x_a] \cdot [\mathbf{k}_{c,1}]\\
[\mathbf{v}_{c,x_d}] &= (1-[x_d]) \cdot [\mathbf{k}_{c,0}] + [x_a] \cdot [\mathbf{k}_{c,1}]
\end{aligned}$$

In each equation either the value $\mathbf{k}_{c,0}$ or the value $\mathbf{k}_{c,1}$ is taken, depending on the corresponding indicator value. Once again, these can be computed using **Add** and **Multiply**.

(c) **Calculate garbled labels:** Party i knows the value of $k^i_{w,b}$ (for wire w that enters gate g) for $b \in \{0,1\}$, and so can compute the $2 \cdot n$ values $F_{k^i_{w,b}}(0\,\|\,1\,\|\,g), \ldots, F_{k^i_{w,b}}(0\,\|\,n\,\|\,g)$ and $F_{k^i_{w,b}}(1\,\|\,1\,\|\,g), \ldots, F_{k^i_{w,b}}(1\,\|\,n\,\|\,g)$. Party i inputs them by calling **InputData** on the functionality $\mathcal{F}_{\mathsf{SPDZ}}$. The resulting input pseudorandom vectors are denoted by

$$\begin{aligned}
[F^0_{k^i_{w,b}}(g)] &= [F_{k^i_{w,b}}(0\,\|\,1\,\|\,g), \ldots, F_{k^i_{w,b}}(0\,\|\,n\,\|\,g)]\\
[F^1_{k^i_{w,b}}(g)] &= [F_{k^i_{w,b}}(1\,\|\,1\,\|\,g), \ldots, F_{k^i_{w,b}}(1\,\|\,n\,\|\,g)].
\end{aligned}$$

The parties now compute $[\mathbf{A}_g], [\mathbf{B}_g], [\mathbf{C}_g], [\mathbf{D}_g]$, using **Add**, via

$$\begin{aligned}
[\mathbf{A}_g] &= \sum\nolimits_{i=1}^{n} \left([F^0_{k^i_{a,0}}(g)] + [F^0_{k^i_{b,0}}(g)]\right) \;+\; [\mathbf{v}_{c,x_a}]\\
[\mathbf{B}_g] &= \sum\nolimits_{i=1}^{n} \left([F^1_{k^i_{a,0}}(g)] + [F^0_{k^i_{b,1}}(g)]\right) \;+\; [\mathbf{v}_{c,x_b}]\\
[\mathbf{C}_g] &= \sum\nolimits_{i=1}^{n} \left([F^0_{k^i_{a,1}}(g)] + [F^1_{k^i_{b,0}}(g)]\right) \;+\; [\mathbf{v}_{c,x_c}]\\
[\mathbf{D}_g] &= \sum\nolimits_{i=1}^{n} \left([F^1_{k^i_{a,1}}(g)] + [F^1_{k^i_{b,1}}(g)]\right) \;+\; [\mathbf{v}_{c,x_d}]
\end{aligned}$$

where every $+$ operation is performed on vectors of n elements.

4. **Notify parties:** Output construction-done.

The functions f_g in Step 3a above depend on the specific gate being evaluated. For example, on clear values we have,

– If $f_g = \wedge$ (i.e. the AND function), $\lambda_a = 1$, $\lambda_b = 1$ and $\lambda_c = 0$ then $x_a = ((1 \wedge 1) - 0)^2 = (1 - 0)^2 = 1$. Similarly $x_b = ((1 \wedge (1-1)) - 0)^2 = (0-0)^2 = 0$, $x_c = 0$ and $x_d = 0$. The parties can compute f_g on shared values $[x]$ and $[y]$ by computing $f_g([x],[y]) = [x] \cdot [y]$.
– If $f_g = \oplus$ (i.e. the XOR function), then $x_a = ((1 \oplus 1) - 0)^2 = (0-0)^2 = 0$, $x_b = ((1 \oplus (1-1)) - 0)^2 = (1-0)^2 = 1$, $x_c = 1$ and $x_d = 0$. The parties can compute f_g on shared values $[x]$ and $[y]$ by computing $f_g([x],[y]) = [x] + [y] - 2 \cdot [x] \cdot [y]$.

Below, we will show how $[x_a]$, $[x_b]$, $[x_c]$ and $[x_d]$ can be computed more efficiently.

3.3 Circuit Complexity

In this section we analyze the complexity of the above circuit in terms of the number of multiplication gates and its depth. We are highly concerned with multiplication gates since, given the SPDZ shares $[a]$ and $[b]$ of the secrets a, and b resp., an interaction between the parties is required to achieve a secret sharing of the secret $a \cdot b$. Achieving a secret sharing of a linear combination of a and b (i.e. $\alpha \cdot a + \beta \cdot b$ where α and β are constants), however, can be done locally and is thus considered negligible. We are interested in the depth of the circuit because it gives a lower bound on the number of rounds of interaction that our circuit requires (note that here, as before, we are concerned with the depth in terms of multiplication gates).

Multiplication gates: We first analyze the number of multiplication operations that are carried out per gate (i.e. in step 3) and later analyze the entire circuit.

– **Multiplications per gate.** We will follow the calculation that is done per gate chronologically as it occurs in step 3 of preprocessing-II phase:
  1. In order to calculate the indicators in step 3a it suffices to compute one multiplication and 4 squares. We can do this by altering the equations a little. For example, for $f_g = AND$, we calculate the indicators by first computing $[t] = [\lambda_a] \cdot [\lambda_b]$ (this is the only multiplication) and then $[x_a] = ([t] - [\lambda_c])^2$, $[x_b] = ([\lambda_a] - [t] - [\lambda_c])^2$, $[x_c] = ([\lambda_b] - [t] - [\lambda_c])^2$, and $[x_d] = (1 - [\lambda_a] - [\lambda_b] + [t] - [\lambda_c])^2$.
$$\begin{aligned}[x_a] &= ([t] - [\lambda_c])^2\\ [x_b] &= ([\lambda_a] - [t] - [\lambda_c])^2\\ [x_c] &= ([\lambda_b] - [t] - [\lambda_c])^2\\ [x_d] &= (1 - [\lambda_a] - [\lambda_b] + [t] - [\lambda_c])^2\end{aligned}$$
     As another example, for $f_g = XOR$, we first compute $[t] = [\lambda_a] \oplus [\lambda_b] = [\lambda_a] + [\lambda_b] - 2 \cdot [\lambda_a] \cdot [\lambda_b]$ (this is the only multiplication), and then

$[x_a] = ([t] - [\lambda_c])^2$, $[x_b] = (1 - [\lambda_a] - [\lambda_b] + 2 \cdot [t] - [\lambda_c])^2$, $[x_c] = [x_b]$, and $[x_d] = [x_a]$.

$$
\begin{aligned}
[x_a] &= ([t] - [\lambda_c])^2 \\
[x_b] &= (1 - [\lambda_a] - [\lambda_b] + 2 \cdot [t] - [\lambda_c])^2 \\
[x_c] &= [x_b] \\
[x_d] &= [x_a]
\end{aligned}
$$

Observe that in XOR gates only two squaring operations are needed.

2. To obtain the correct vector (in step 3b) which is used in each garbled label, we carry out 8 multiplications. Note that in XOR gates only 4 multiplications are needed, because $\mathbf{k}_{c,x_c} = \mathbf{k}_{c,x_b}$ and $\mathbf{k}_{c,x_d} = \mathbf{k}_{c,x_a}$.

Summing up, we have 4 squaring operations in addition to 9 multiplication operations per AND gate and 2 squarings in addition to 5 multiplications per XOR gate.

– **Multiplications in the entire circuit.** Denote the number of multiplication operation per gate (i.e. 13 for AND and 7 for XOR) by c, we get $G \cdot c$ multiplications for garbling all gates (where G is the number of gates in the boolean circuit computing the functionality f). Besides garbling the gates we have no other multiplication operations in the circuit. Thus we require $c \cdot G$ multiplications in total.

Depth of the circuit and round complexity: Each gate can be garbled by a circuit of depth 3 (two levels are required for step 3a and another one for step 3b). Recall that additions are local operations only and thus we measure depth in terms of multiplication gates only. Since all gates can be garbled in parallel this implies an overall depth of three. (Of course in practice it may be more efficient to garble a set of gates at a time so as to maximize the use of bandwidth and CPU resources.) Since the number of rounds of the SPDZ protocol is in the order of the depth of the circuit, it follows that $\mathcal{F}_{\mathsf{offline}}$ can be securely computed in a constant number of rounds.

Other Considerations: The overall cost of the pre-processing does not just depend on the number of multiplications. Rather, the parties also need to produce the random data via calls to **Random** and **RandomBit** to the functionality $\mathcal{F}_{\mathsf{SPDZ}}$.[4] It is clear all of these can be executed in parallel. If W is the number of wires in the circuit then the total number of calls to **RandomBit** is equal to W, whereas the total number of calls to **Random** is $2 \cdot n \cdot W$.

Arithmetic vs Boolean Circuits: Our protocol will perform favourably for functions which are reasonably represented as boolean circuit, but the low round complexity may be outweighed by other factors when the function can be expressed much more succinctly using an arithmetic circuit, or other programatic

[4] These **Random** calls are followed immediately with an **Open** to a party. However, in SPDZ **Random** followed by **Open** has roughly the same cost as **Random** alone.

representation as in [12]. In such cases, the performance would need to be tested for the specific function.

3.4 Expected Runtimes

To estimate the running time of our protocol, we extrapolate from known public data [8, 7]. The offline phase of our protocol runs both the offline and online phases of the SPDZ protocol. The numbers below refer to the SPDZ offline phase, as described in [7], with covert security and a 20% probability of cheating, using finite fields of size 128-bits, to obtain the following generation times (in milli-seconds). As described in [7], comparable times are obtainable for running in the fully malicious mode (but more memory is needed).

No. Parties	Beaver Triple	RandomBit	Random	Input
2	0.4	0.4	0.3	0.3
3	0.6	0.5	0.4	0.4
4	0.9	1.2	0.9	0.9

Table 1. SPDZ offline generation times in milliseconds per operation

The implementation of the SPDZ online phase, described in both [7] and [12], reports online throughputs of between 200,000 and 600,000 per second for multiplication, depending on the system configuration. As remarked earlier the online time of other operations is negligible and are therefore ignored.

To see what this would imply in practice consider the AES circuit described in [17]; which has become the standard benchmarking case for secure computation calculations. The basic AES circuit has around 33,000 gates and a similar number of wires, including the key expansion within the circuit.[5] Assuming the parties share a XOR sharing of the AES key, (which adds an additional $2 \cdot n \cdot 128$ gates and wires to the circuit), the parameters for the **Initialize** call to the $\mathcal{F}_{\mathsf{SPDZ}}$ functionality in the preprocessing-I protocol will be

$$M \approx 429,000, \quad B \approx 33,000, \quad R \approx 66,000 \cdot n, \quad I \approx 66,000 \cdot n + 128.$$

Using the above execution times for the SPDZ protocol we can then estimate the time needed for the two parts of our processing step for the AES circuit. The expected execution times, in seconds, are given in the following table. These expected times, due to the methodology of our protocol, are likely to estimate both the latency and throughput amortized over many executions.

No. Parties	preprocessing-I	preprocessing-II
2	264	0.7–2.0
3	432	0.7–2.0
4	901	0.7–2.0

[5] Note that unlike [17] and other Yao based techniques we cannot process XOR gates for free. On the other hand we are not restricted to only two parties.

The execution of the online phase of our protocol, when the parties are given their inputs and actually want to compute the function, is very efficient: all that is needed is the evaluation of a garbled circuit based on the data obtained in the offline stage. Specifically, for each gate each party needs to process two input wires, and for each wire it needs to expand n seeds to a length which is n times their original length (where n denotes the number of parties). Namely, for each gate each party needs to compute a pseudorandom function $2n^2$ times (more specifically, it needs to run $2n$ key schedulings, and use each key for n encryptions). We examined the cost of implementing these operations for an AES circuit of 33,000 gates when the pseudorandom function is computed using the AES-NI instruction set. The run times for $n = 2, 3, 4$ parties were 6.35msec, 9.88msec and 15msec, respectively, for C code compiled using the gcc compiler on a 2.9GHZ Xeon machine. The actual run time, including all non-cryptographic operations, should be higher, but of the same order.

Our run-times estimates compare favourably to several other results on implementing secure computation of AES in a multiparty setting:

- In [6] an actively secure computation of AES using SPDZ took an offline time of over five minutes per AES block, with an online time of around a quarter of a second; that computation used a security parameter of 64 as opposed to our estimates using a security parameter of 128.
- In [12] another experiment was shown which can achieve a latency of 50 milliseconds in the online phase for AES (but no offline times are given).
- In [16] the authors report on a two-party MPC evaluation of the AES circuit using the Tiny-OT protocol; they obtain for 80 bits of security an amortized offline time of nearly three seconds per AES block, and an amortized online time of 30 milliseconds; but the reported non-amortized latency is much worse. Furthermore, this implementation is limited to the case of *two parties*, whereas we obtain security for multiple parties.

Most importantly, all of the above experiments were carried out in a LAN setting where communication latency is very small. However, in other settings where parties are not connect by very fast connections, the effect of the number of rounds on the protocol will be extremely significant. For example, in [6], an arithmetic circuit for AES is constructed of depth 120, and this is then reduced to depth 50 using a bit decomposition technique. Note that if parties are in separate geographical locations, then this number of rounds will very quickly dominate the running time. For example, the latency on Amazon EC2 between Virginia and Ireland is 75ms. For a circuit depth of 50, and even assuming just a *single* round per level, the running-time cannot be less than 3750 milliseconds (even if computation takes *zero time*). In contrast, our online phase has just 2 rounds of communication and so will take in the range of 150 milliseconds. We stress that even on a much faster network with latency of just 10ms, protocols with 50 rounds of communication will still be slow.

Acknowledgments

The first and fourth authors were supported in part by the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC consolidators grant agreement n. 615172 (HIPS). The second author was supported under the European Union's Seventh Framework Program (FP7/2007-2013) grant agreement n. 609611 (PRACTICE), and by a grant from the Israel Ministry of Science, Technology and Space (grant 3-10883). The third author was supported in part by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO and by EPSRC via grant EP/I03126X. The first and third authors were also supported by an award from EPSRC (grant EP/M012824), from the Ministry of Science, Technology and Space, Israel, and the UK Research Initiative in Cyber Security.

References

1. D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In H. Ortiz, editor, *22nd STOC*, pages 503–513. ACM, 1990.
2. A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In P. Ning, P. F. Syverson, and S. Jha, editors, *ACM CCS*, pages 257–266. ACM, 2008.
3. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In Simon [20], pages 1–10.
4. D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In Simon [20], pages 11–19.
5. S. G. Choi, J. Katz, A. J. Malozemoff, and V. Zikas. Efficient three-party computation from cut-and-choose. In Garay and Gennaro [9], pages 513–530.
6. I. Damgård, M. Keller, E. Larraia, C. Miles, and N. P. Smart. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In I. Visconti and R. D. Prisco, editors, *SCN 2012*, volume 7485 of *LNCS*, pages 241–263. Springer, 2012.
7. I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In J. Crampton, S. Jajodia, and K. Mayes, editors, *ESORICS*, volume 8134 of *LNCS*, pages 1–18. Springer, 2013.
8. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [19], pages 643–662.
9. J. A. Garay and R. Gennaro, editors. *Advances in Cryptology - CRYPTO 2014*, volume 8617 of *LNCS*. Springer, 2014.
10. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In A. V. Aho, editor, *Proceedings STOC 1987*, pages 218–229. ACM, 1987.
11. V. Goyal. Constant round non-malleable protocols using one way functions. In L. Fortnow and S. P. Vadhan, editors, *Proceedings STOC 2011*, pages 695–704. ACM, 2011.
12. M. Keller, P. Scholl, and N. P. Smart. An architecture for practical actively secure MPC with dishonest majority. In A. Sadeghi, V. D. Gligor, and M. Yung, editors, *2013 ACM CCS '13*, pages 549–560. ACM, 2013.

13. E. Larraia, E. Orsini, and N. P. Smart. Dishonest majority multi-party computation for binary circuits. In Garay and Gennaro [9], pages 495–512.
14. Y. Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology - CRYPTO 2013*, volume 8043 of *LNCS*, pages 1–17. Springer, 2013.
15. Y. Lindell and B. Riva. Cut-and-choose yao-based secure computation in the online/offline and batch settings. In Garay and Gennaro [9], pages 476–494.
16. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [19], pages 681–700.
17. B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In M. Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.
18. T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In D. S. Johnson, editor, *Proceedings STOC 1989*, pages 73–85. ACM, 1989.
19. R. Safavi-Naini and R. Canetti, editors. *Advances in Cryptology - CRYPTO 2012*, volume 7417 of *LNCS*. Springer, 2012.
20. J. Simon, editor. *Proceedings STOC 1988*. ACM, 1988.
21. A. C. Yao. Protocols for secure computations. In *Proceedings FOCS 1982*, pages 160–164. IEEE Computer Society, 1982.

A The BMR Protocol of [1]

In this appendix we outline the basis of our protocol, which is the BMR protocol of Beaver, Micali and Rogaway for semi-honest adversaries. (BMR also have a version for malicious adversaries. However, it requires an honest majority and is also not concretely efficient.) The protocol is comprised of an offline-phase and an online-phase. During the offline-phase the garbled circuit is created by the players, while in the online-phase a matching set of garbled inputs is exchanged between the players and each of them evaluates the garbled circuit locally. The protocol is based on the following data items:

Seeds and superseeds: Two random seeds are associated with each wire in the circuit by each player. We denote the 0-seed and 1-seed that are chosen by player P_i (where $1 \leq i \leq n$) for wire w as $s^i_{w,0}$ and $s^i_{w,1}$ (where $0 \leq w < W$ and W is the number of wires in the circuit and $s^i_{w,j} \in \{0,1\}^\kappa$). During the garbling process the players produce two *superseeds* for each wire, where the 0-superseed and 1-superseed for wire w are a simple concatenation of the 0-seeds and 1-seeds chosen by all the players, namely, $S_{w,0} = s^1_{w,0}\| \cdots \|s^n_{w,0}$ and $S_{w,1} = s^1_{w,1}\| \cdots \|s^n_{w,1}$ where $\|$ denotes concatenation. Note that $S_{w,j} \in \{0,1\}^L$ where $L = n \cdot \kappa$.

Garbling wire values: For each gate g which calculates the function f_g (where $f_g : \{0,1\} \times \{0,1\} \to \{0,1\}$), the garbled gate of g is computed such that the superseeds associated with the output wire are encrypted (via a simple XOR) using the superseeds associated with the input wires, according to the truth table of f_g. Specifically, a superseed $S_{w,0} = s^1_{w,0}\| \cdots \|s^n_{w,0}$ is used to encrypt a value

M of length L by computing $M \bigoplus_{i=1}^{n} G(s_{w,0}^{i})$, where G is a pseudo-random generator stretching a seed of length κ to an output of length L. This means that *every* one of the seeds that make up the superseed must be known in order to learn the mask and decrypt.

Masking values: Using random seeds instead of the original 0/1 values does not hide the original value if it is known that the first seed corresponds to 0 and the second seed to 1. Therefore, an unknown random *masking bit*, denoted by λ_w, is assigned to wire w (for $0 \leq w < W$). These masking bits remain unknown to the players during the entire protocol, thereby preventing them from knowing the *real values* ρ_w that pass through the wires. The values that the players *do* know are called the *external values* Λ_w. An external value is defined to be the exclusive-or of the real value and the masking value; i.e., $\Lambda_w = \rho_w \oplus \lambda_w$. When evaluating the garbled circuit the players only see the external values of the wires, which are random bits that tell nothing about the real values, unless they know the masking values. We remark that each party P_i is given the masking value associated with its input. Thus, it can compute the external value itself (based on its actual input) and can send it to all other parties.

BMR garbled gates and circuit: We can now define the BMR garbled circuit, which consists of the set of garbled gates, where a garbled gate is defined via a functionality that maps inputs to outputs. Let g be a gate with input wires a, b and output wire c. Each party P_i (for $1 \leq i \leq n$) inputs the seeds $s_{a,0}^{i}, s_{a,1}^{i}, s_{b,0}^{i}, s_{b,1}^{i}, s_{c,0}^{i}, s_{c,1}^{i}$. Thus, the superseeds produced are $S_{a,0}, S_{a,1}, S_{b,0}, S_{b,1}, S_{c,0}, S_{c,1}$, where each superseed is given by $S_{\alpha,\beta} = s_{\alpha,\beta}^{1} \| \cdots \| s_{\alpha,\beta}^{n}$. In addition, P_i also inputs the output of a pseudo-random generator G applied on each of these seeds, and its shares of the masking bits, i.e. $\lambda_a^i, \lambda_b^i, \lambda_c^i$.

The output is the garbled gate of g which comprised of a table of four *ciphertexts*, each of them encrypting either $S_{c,0}$ or $S_{c,1}$. The property of the gate construction is that given one superseed for a and one superseed for b it is possible to to decrypt exactly one ciphertext, and reveal the appropriate superseed for c (based on the values on the input wires and the gate type). The functionality, garble-gate-BMR, for garbling a single gate, is formally described in Functionality 5.

The BMR Online Phase: In the online-phase the players only have to obtain one superseed for every circuit-input wire, and then every player can evaluate the circuit on his own, without interaction with the rest of the players. Formally, the protocol that implements the online phase is given by Protocol 2.

Functionality 5 (garble-gate-BMR)

Let κ denote the security parameter, and let $G : \{0,1\}^\kappa \to \{0,1\}^{2n\kappa}$ be a pseudo-random generator. Denote the first $L = n \cdot \kappa$ bits of the output of G by G^1, and the last $n\kappa$ bits of the output of G by G^2.
The garbling of gate g computing $f_g : \{0,1\} \times \{0,1\} \to \{0,1\}$ with inputs wires a, b and output wire c is defined as follows:

Inputs: For each gate the inputs are given by

1. **Seeds:** $s^1_{a,0}, \ldots, s^n_{a,0}$, $s^1_{a,1}, \ldots, s^n_{a,1}$, $s^1_{b,0}, \ldots, s^n_{b,0}$, $s^1_{b,1}, \ldots, s^n_{b,1}$, $s^1_{c,0}, \ldots, s^n_{c,0}$, $s^1_{c,1}, \ldots, s^n_{c,1}$ where each seed is in $\{0,1\}^\kappa$.
2. **PRG output:** The output of G applied to each of the seeds above, such that the first $n \cdot \kappa$ bits of the output are denoted by G^1 and the other $n \cdot \kappa$ bits by G^2.
3. **Masking bits.** Bits λ_a, λ_b and λ_c.

Outputs: The garbled gate of g is the following four ciphertexts A_g, B_g, C_g, D_g (in this order that is determined by the external values):

$$\begin{aligned}
A_g &= G^1(s^1_{a,0}) \oplus \cdots \oplus G^1(s^n_{a,0}) \oplus G^1(s^1_{b,0}) \oplus \cdots \oplus G^1(s^n_{b,0}) \\
&\qquad \oplus \begin{cases} S_{c,0} & \text{if } f_g(\lambda_a, \lambda_b) = \lambda_c \\ S_{c,1} & \text{otherwise} \end{cases} \\
B_g &= G^2(s^1_{a,0}) \oplus \cdots \oplus G^2(s^n_{a,0}) \oplus G^1(s^1_{b,1}) \oplus \cdots \oplus G^1(s^n_{b,1}) \\
&\qquad \oplus \begin{cases} S_{c,0} & \text{if } f_g(\lambda_a, \bar{\lambda_b}) = \lambda_c \\ S_{c,1} & \text{otherwise} \end{cases} \\
C_g &= G^1(s^1_{a,1}) \oplus \cdots \oplus G^1(s^n_{a,1}) \oplus G^2(s^1_{b,0}) \oplus \cdots \oplus G^2(s^n_{b,0}) \\
&\qquad \oplus \begin{cases} S_{c,0} & \text{if } f_g(\bar{\lambda_a}, \lambda_b) = \lambda_c \\ S_{c,1} & \text{otherwise} \end{cases} \\
D_g &= G^2(s^1_{a,1}) \oplus \cdots \oplus G^2(s^n_{a,1}) \oplus G^2(s^1_{b,1}) \oplus \cdots \oplus G^2(s^n_{b,1}) \\
&\qquad \oplus \begin{cases} S_{c,0} & \text{if } f_g(\bar{\lambda_a}, \bar{\lambda_b}) = \lambda_c \\ S_{c,1} & \text{otherwise} \end{cases}
\end{aligned}$$

Protocol 2 (Protocol BMR-online-phase)

Step 1 – send values:

1. Every player P_i broadcasts the external value values on the wires associated with its input. At the end of this step the players know the external value Λ_w for every circuit-input wire w. (Recall that P_i knows λ_w and so can compute Λ_w based on its input.)
2. Every player P_i broadcasts one seed for each circuit-input wire, namely, the Λ_w-seed. At the end of this step the players know the Λ_w-superseed for every circuit-input wire.

Step 1 – evaluate circuit: The players evaluate the circuit from bottom up, such that to obtain the superseed of an output wire of the gate, use A_g if the external values of g's input wires are $\Lambda_a, \Lambda_b = (0, 0)$, use B_g if $\Lambda_a, \Lambda_b = (0, 1)$, C_g if $\Lambda_a, \Lambda_b = (1, 0)$ and D_g if $\Lambda_a, \Lambda_b = (1, 1)$ where a, b are the input wires. (see the original paper for more details).

Correctness: We explain now why the conditions for masking $S_{c,0}$ and $S_{c,1}$ are correct. The external values Λ_a, Λ_b indicate to the parties which ciphertext to decrypt. Specifically, the parties decrypt A_g if $\Lambda_a = \Lambda_b = 0$, they decrypt B_g if $\Lambda_a = 0$ and $\Lambda_b = 1$, they decrypt C_g if $\Lambda_a = 1$ and $\Lambda_b = 0$, and they decrypt D_g if $\Lambda_a = \Lambda_b = 1$.

We need to show that given S_{a,Λ_a} and S_{b,Λ_b}, the parties obtain S_{c,Λ_c}. Consider the case that $\Lambda_a = \Lambda_b = 0$ (note that $\Lambda_a = 0$ means that $\lambda_a = \rho_a$, and $\Lambda_a = 1$ means that $\lambda_a \neq \rho_a$, where ρ_a is the real value). Since $\rho_a = \lambda_a$ and $\rho_b = \lambda_b$ we have that $f_g(\lambda_a, \lambda_b) = f_g(\rho_a, \rho_b)$. If $f_g(\lambda_a, \lambda_b) = \lambda_c$ then by definition $f_g(\rho_a, \rho_b) = \rho_c$, and so we have $\lambda_c = \rho_c$ and thus $\Lambda_c = 0$. Thus, the parties obtain $S_{c,0} = S_{c,\Lambda_c}$. In contrast, if $f_g(\lambda_a, \lambda_b) \neq \lambda_c$ then by definition $f_g(\rho_a, \rho_b) \neq \rho_c$, and so we have $\lambda_c = \bar{\rho_c}$ and thus $\Lambda_c = 1$. A similar analysis show that the correct values are encrypted for all other combinations of Λ_a, Λ_b.

B A Generic Protocol to Implement $\mathcal{F}_{\mathsf{offline}}$

In this Appendix we give a generic protocol Π_{offline} which implements $\mathcal{F}_{\mathsf{offline}}$ using any protocol which implements the generic MPC functionality $\mathcal{F}_{\mathsf{MPC}}$. The protocol is very similar to the protocol in the main body which is based on the SPDZ protocol, however this generic functionality requires more rounds of communication (but still requires constant rounds). Phase Two is implemented exactly as in Section 3, so the only change we need is to alter the implementation of Phase One; which is implemented as follows:

1. **Initialize the MPC Engine:** Call **Initialize** on the functionality $\mathcal{F}_{\mathsf{MPC}}$ with input p, a prime with $2^\kappa < p < 2^{\kappa+1}$.
2. **Generate wire masks:** For every circuit wire w we need to generate a sharing of the (secret) masking-values λ_w. Thus for *all* wires w the players execute the following commands

– Player i calls **InputData** on the functionality $\mathcal{F}_{\mathsf{MPC}}$ for a random value λ_w^i of his choosing.
– The players compute

$$[\mu_w] = \prod_{i=1}^{n} [\lambda_w^i],$$
$$[\lambda_w] = \frac{[\mu_w] + 1}{2},$$
$$[\tau_w] = [\mu_w] \cdot [\mu_w] - 1.$$

– The players open $[\tau_w]$ and if $\tau_w \neq 0$ for any wire w they abort.

3. **Generate garbled wire values:** For every wire w, each party $i \in [1, \ldots, n]$ and for $j \in \{0, 1\}$, player i generates a random value $k_{w,j}^i \in \mathbb{F}_p$ and call **InputData** on the functionality $\mathcal{F}_{\mathsf{MPC}}$ so as to obtain $[k_{w,j}^i]$. The vector of shares $[k_{w,j}^i]_{i=1}^n$ we shall denote by $[\mathbf{k}_{w,j}]$.

C Security Proof

The security proof is demonstrated by two steps. In the first step we reduce security in the semi-honest case, i.e. for an adversary $\mathcal{A}$ that does not deviate from the described protocol and only tries to learn information from the transcript, to the security of the original BMR protocol. In the second step we show that our protocol remains secure even if $\mathcal{A}$ is *malicious*, i.e. is allowed to deviate from the protocol. This second step is performed by giving a reduction from the malicious to the semi-honest model. In both steps the adversary $\mathcal{A}$ is assumed to corrupt parties in the beginning of the execution of our protocol.

To be able to follow the proof smoothly we first present some conventions and notations. In both the original BMR protocol and our protocol the players obtain a garbled circuit and a matched set of garbled inputs, they are then able to evaluate the circuit without further interaction. The players evaluate the circuit from bottom up until they reach the circuit-output wires, i.e. the input wires are said to be at the "bottom" of the circuit, whilst the output wires are at the "top". In their evaluation the players use the garbled gate of gate g to reveal a single external value for wire c (i.e. Λ_c, where c is g's output wire) together with an appropriate key-vector $\mathbf{k}_{c,\Lambda_c} = k_{c,\Lambda_c}^1, \ldots, k_{c,\Lambda_c}^n$. There is only one entry in the garbled gate that can be used to reveal the pair $(\Lambda_c, \mathbf{k}_{c,\Lambda_c})$; specifically if g's input wires are a and b then the $(2\Lambda_a + \Lambda_b)$-th entry in the table of the garbled gate of g is used (where the entries indices are 0 for A_g, 1 for B_g, 2 for C_g and 3 for D_g). For each gate we call the garbled gate's entry for which the players evaluate that gate as the *active entry* and the other three entries as *inactive* entries. Similarly we use the term *active signal* to denote the value Λ_c that is revealed for some wire c, and the term *active path* for the set of active signals that have been revealed to the players during the evaluation of the circuit. Recall that in the online phase of our protocol the players exchange the active signal of all the circuit-input wires. We denote by I the set of indices of the players that are under the control of the adversary $\mathcal{A}$, and by x_I their inputs to the functionality (note that in the malicious case these inputs might be different from the inputs that the players have been given originally). In the same manner, J is the set of indices of the honest-parties and x_J their inputs such that $|I \cup J| = n$ and $I \cap J = \varnothing$. We denote by W, W_{in} and W_{out} the sets of all wires, the set of circuit-input wires (a.k.a. attached wires) and the set of circuit-output wires of the circuit C. We denote the set of gates in the circuit as $G = \{g_1, \ldots, g_{|G|}\}$. Recall that κ is the security parameter.

C.1 Security in the semi-honest model

View 1 (The view $\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}}$)

For every $i \in I$ the adversary sees the following:

1. **Masking shares:** Shares of the masking values for all wires W, i.e. $\{\lambda_w^i \in \{0,1\} \mid w \in W\}$.
2. **Masking values for attached wires:** The ℓ masking values λ_w of P_i's attached wires w are revealed in the clear.
3. **Seeds:** Player P_i's seed values $\{s_{w,0}^i, s_{w,1}^i \in \{0,1\}^\kappa \mid w \in W\}$.
4. **Seed extensions:** For each seed $s_{w,b}^i$ player P_i sees two pseudo-random extensions $G^1(s_{w,b}^i), G^2(s_{w,b}^i) \in \{0,1\}^{n\kappa}$.

In addition the adversary sees:

1. **Masking values for output wires:** The masking values $\{\lambda_w \in \{0,1\} \mid w \in W_{out}\}$.
2. **Garbled circuit:** For every gate g the garbled table $\{A_g, B_g, C_g, D_g \mid g \in G\}$ where $A_g, B_g, C_g, D_g \in \{0,1\}^{n\kappa}$.
3. **Inputs:** The input values $\bar{x}_I$.
4. **Active path:** For every wire w in the circuit one active signal together with its matched superseed, i.e. $(\Lambda_w, S_{w,\Lambda_w})$, using one entry of the garbled gate. The rest of the values (i.e. the inactive entries) are indistinguishable from random.

The idea is to show that there exist a probabilistic polynomial-time procedure, $\mathcal{P}$, whose input is a view sampled from the view distribution of a semi-honest adversary involved in a real execution of the original BMR protocol[6], namely $\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}}$ in View 1; and its output is a view from the view distribution of a semi honest adversary involved in a real execution of our protocol, namely $\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}$ in View 2. Formally, the procedure is defined as

$$\mathcal{P} : \{\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}}\}_{\bar{x}} \to \{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}\}_{\bar{x}}$$

[6] In this section we actually mean to the execution in the hybrid model where the parties have access to the underlying MPC functionality, we call it as *real* execution for convenience.

where $\bar{x} = x_1, \ldots, x_n$ is the players' input to the functionality.

In this section we present the procedure $\mathcal{P}$ and show that $\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{\bar{x}}$ and $\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}\}_{\bar{x}}$ are indistinguishable. We then show that the existence of a simulator, $\mathcal{S}_{\text{BMR}}$, for $\mathcal{A}$'s view in the execution of the original BMR protocol implies the existence of a simulator $\mathcal{S}_{\text{OUR}}$ for $\mathcal{A}$'s view in the execution of our protocol. In the following we first describe $\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}}$ (View 1) and $\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}$ (View 2), then we describe the procedure $\mathcal{P}$ and prove the mentioned claims.

View 2 (The view $\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}$)

For every $i \in I$ the adversary sees the following:

1. **Masking values for attached wires:** The ℓ masking values λ_w of P_i's attached wires w are revealed in the clear.
2. **Keys.** Player P_i's random keys $\{k_{w,0}^i, k_{w,1}^i \in \mathbb{F}_p \mid w \in W\}$.
3. **Keys extensions.** For every key $k_{w,b}^i$, and for every gate g which wire w enters into, the values

$$\Big\{F_{k_{w,b}^i}(0 \,\|\, 1 \,\|\, g), \ldots, F_{k_{w,b}^i}(0 \,\|\, n \,\|\, g),$$
$$F_{k_{w,b}^i}(1 \,\|\, 1 \,\|\, g), \ldots, F_{k_{w,b}^i}(1 \,\|\, n \,\|\, g) \mid w \in W\Big\}.$$

In addition the adversary sees:

1. **Masking values for output wires:** The masking values $\{\lambda_w \in \{0,1\} \mid w \in W_{out}\}$.
2. **Construction done.** The message construction-done broadcasted by the functionality.
3. **Inputs.** The input values $\bar{x}_I$.
4. **Open message** The message open.
5. **Garbled circuit.** For every gate g $\{A_g, B_g, C_g, D_g \mid g \in G\}$ where $A_g, B_g, C_g, D_g \in (\mathbb{F}_p)^n$.
6. **Active path.** For every wire w in the circuit one active signal together with its matched key-vector, i.e. $(\Lambda_w, \mathbf{k}_{w,\Lambda_w})$, using one entry of the garbled gate.

We are ready to describe the procedure $\mathcal{P}$ (Procedure 3), which is given a view $\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}}$ that is sampled from the distribution of the adversary's views under the input $\bar{x}$ of the players in the original BMR protocol, and outputs a view from the distribution of the adversary's views in our protocol (i.e. $\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}$). We will then show that the resulting distribution of views is indistinguishable from $\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}$ for every $\bar{x}$. Since $\mathcal{P}$ sees the garbled circuit and the matched set of (garbled) inputs from all players, it can evaluate the circuit by itself and determine the active path and the output $\bar{y}_I$, however, $\mathcal{P}$ does not knows $\bar{x}_J$

(it only knows $\bar{x}_I$) and thus cannot construct a garbled circuit for our protocol from scratch, it must instead use the information that can be extracted from it's input view.

Procedure 3 (The Procedure $\mathcal{P}$)

Input. A view v taken from distribution $\text{VIEW}_{\mathcal{A}}^{\mathsf{BMR}}$ under the input $\bar{x}$.
Output. A view v' conforming to the message flow in $\text{VIEW}_{\mathcal{A}}^{\mathsf{Our}}$.
The procedure proceeds as follows:

1. Take the masking values for the attached wires and for the output wires W_{out} to be the same as in v.
2. Set x_I to be the same as in v.
3. To construct the garbled circuit:
   (a) Choose a random set of keys $\{k_{w,b}^i \mid w \in W, b \in \{0,1\}, i \in I \cup J\}$ for the players, and for each key compute the appropriate $2n$ PRF values.
   (b) Choose a random set of masking values for all wires that are not attached with the players P_I and are not in W_{out}.
   (c) For every gate g in the circuit, with input wires a, b and output wire c, the algorithm sets the the garbled entries (except one as described immediately) to be random values from $(\mathbb{F}_p)^n$ whilst for the $(2\cdot\Lambda_a+\Lambda_b)$-th entry the algorithm instead conceals the Λ_c key-vector (in contrast to the real construction in which the key-vector that the entry conceals depends on the masking values of a, b and c). That is, when the algorithm construct the garbled gates it ignores the masking values that it chose in the previous step. For example, take $\Lambda_a = 1$,$\Lambda_b = 0$ and $\Lambda_c = 1$ then the entry by which the players evaluate the gate is the $2 \cdot \Lambda_a + \Lambda_b = 2$ (i.e. the third) entry which is C_g. Thus $\mathcal{P}$ makes C_g to encrypt the 1-key vector, i.e. $\mathbf{k}_{c,1}$ by:

$$A_g^j \xleftarrow{R} \mathbb{F}_p$$
$$B_g^j \xleftarrow{R} \mathbb{F}_p$$
$$C_g^j = \left(\sum_{i=1}^{n} F_{k_{a,1}^i}(0\|j\|g) + F_{k_{b,0}^i}(1\|j\|g)\right) + k_{c,1}^j$$
$$D_g^j \xleftarrow{R} \mathbb{F}_p$$

   for $j = 1, \ldots, n$ as described in Functionality 3. Note that we explicitly conceal $k_{c,1}^j$ for every element in $\mathbf{k}_{c,1}$ because we already know from the active path of v that the external value of wire c is $\Lambda_c = 1$.
4. Add the messages construction-done and open to the obvious location in the resulting view.

Claim 4 *Given that the BMR protocol is secure in the semi-honest model, our protocol is secure in the semi-honest model as well.*

Proof. From the security of the BMR protocol we know that

$$\{\mathcal{S}_{\text{BMR}}(1^\kappa, I, x_I, y_I)\}_{\bar{x}} \overset{c}{\equiv} \{\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}}\}_{\bar{x}}$$

thus, for every PPT algorithm, and specifically for algorithm $\mathcal{P}$ it holds that

$$\{\mathcal{P}(\mathcal{S}_{\text{BMR}}(1^\kappa, I, x_I, y_I))\}_{\bar{x}} \overset{c}{\equiv} \{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{\bar{x}}$$

then, if the following computational indistinguishability holds (proven in claim 5)

$$\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}\}_{\bar{x}} \overset{c}{\equiv} \{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{\bar{x}} \tag{1}$$

then by transitivity of indistinguishability, it follows that

$$\begin{aligned} &\{\mathcal{P}(\mathcal{S}_{\text{BMR}}(1^\kappa, I, x_I, y_I))\}_{\bar{x}} \overset{c}{\equiv} \{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{\bar{x}} \overset{c}{\equiv} \{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}\}_{\bar{x}} \\ \Rightarrow &\{\mathcal{P}(\mathcal{S}_{\text{BMR}}(1^\kappa, I, x_I, y_I))\}_{\bar{x}} \qquad \overset{c}{\equiv} \qquad \{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}\}_{\bar{x}} \end{aligned}$$

hence, $\mathcal{P} \circ \mathcal{S}_{\text{BMR}}$ is a good simulator for the view of the adversary in the semi honest model. ■

In the following we prove Equation 1:

Claim 5 *The probability ensemble of the view of the adversary in the real execution of our protocol and the probability ensemble of the view of the adversary resulting by the procedure $\mathcal{P}$, both indexed by the players' inputs to the functionality $\bar{x}$, are computationally indistinguishable. That is:*

$$\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}\}_{\bar{x}} \overset{c}{\equiv} \{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{\bar{x}}$$

Proof. Remember that in the procedure $\mathcal{P}$ we don't have any information about the masking values $\{\lambda w \mid w \in W\}$ (except of those which are known to the adversary), therefore we couldn't compute the indicators x_A, x_B, x_C, x_D (as described in section 3.2) and thus couldn't tell which key vector is encrypted in each entry, that is, we couldn't fill out correctly the four garbled gate's entries A, B, C, D. On the other hand, in the procedure $\mathcal{P}$ we do know the set of external values $\{exvw \mid w \in W\}$, thus, we know for sure that for every gate g, with input wires a, b and output wire c, the key vector encrypted in the $2\Lambda_a + \Lambda_b$-th entry of the garbled table of gate g is the Λ_c -th key vector $\mathbf{k}_{c,\Lambda_c}$.

Let us denote by $\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}\}_{f,\bar{x},k^i_{w,\beta},\lambda j}$ the view of the adversary in the execution of our protocol (which computes the functionality f) with players' inputs $\bar{x}$ when using the keys $\{k^i_{w,\beta} \mid 1 \le i \le n, w \in W, \beta \in \{0,1\}\}$ and the masking values $\{\lambda j \mid j \in W\}$. Similarly, denote by $\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{f,\bar{x},k^i_{w,\beta},\lambda j}$ the view of the adversary in the output of the procedure $\mathcal{P}$.

Given that

$$\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}\}_{f,\bar{x},k^i_{w,\beta},\lambda j} \overset{c}{\equiv} \{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{f,\bar{x},k^i_{w,\beta},\lambda j} \tag{2}$$

are computationally indistinguishable (i.e. under the same functionality, players' inputs, keys and masking values) it follows that

$$\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}\}_{\bar{x}} \overset{c}{\equiv} \{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{\bar{x}}$$

since the functionality, keys and masking values are taken from exactly the same distributions in both cases. In the following (claim 6) we prove equation 2.

Claim 6 *Fix a functionality f, players' inputs $\bar{x}$, keys $\{k_{w,\beta}^i \mid 1 \leq i \leq n, w \in W, \beta \in \{0,1\}\}$ and masking values $\{\lambda j \mid j \in W\}$ used in both the execution of our protocol and the procedure $\mathcal{P}$, then equation (2) holds; that is*

$$\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}\}_{f,\bar{x},k_{w,\beta}^i,\lambda j} \overset{c}{\equiv} \{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{f,\bar{x},k_{w,\beta}^i,\lambda j}$$

Proof. Note that the difference between $\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}\}_{f,\bar{x},k_{w,\beta}^i,\lambda j}$ and $\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{f,\bar{x},k_{w,\beta}^i,\lambda j}$ are the values of the garbled gates' entries which are not in the active path, that is, in $\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}\}_{f,\bar{x},k_{w,\beta}^i,\lambda j}$ these values are computed as described in section 3.2 while in the procedure $\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{f,\bar{x},k_{w,\beta}^i,\lambda j}$ they are just random values from $(\mathbb{F}_p)^n$.

Let $\mathcal{D}$ be a polynomial time distinguisher such that

$$|Pr[\mathcal{D}(\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}\}_{f,\bar{x},k_{w,\beta}^i,\lambda j}) = 1] - Pr[\mathcal{D}(\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{f,\bar{x},k_{w,\beta}^i,\lambda j}) = 1]| = \varepsilon(\kappa)$$

and assume by contradiction that ε is some non-negligible function in κ.

Let C be the boolean circuit that computes the functionality f. Consider C as a set of layers of gates, where the first layer (indexed as layer 1) consists of all gates whose both inputs wires are circuit-input wires, the second layer consists of the gates whose input wires are either from the circuit-input wires or the output wire of some gates from layer 1, and so on. Each gate belongs to the maximal layer possible (e.g. a gate whose input wires are the output wires of two gates, one from layer d_1 and the other from layer d_2, belongs to layer $\max(d_1, d_2) + 1$). We denote the depth of the circuit C (i.e. the maximal layer index) by d.

We define the hybrid H^t as the view in which the gates which belong to layers $1, \ldots, t$ are computed as in the procedure $\mathcal{P}$ (i.e. the inactive entries are just random elements from $(\mathbb{F}_p)^n$) and the gates which belong to the layers $t+1, \ldots, d$ are computed as described in our protocol (section 3.2). Observe that H^0 is distributed exactly as the view of the adversary in $\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}\}_{f,\bar{x},k_{w,\beta}^i,\lambda j}$ and H^d is distributed exactly as the view of the adversary in $\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{f,\bar{x},k_{w,\beta}^i,\lambda j}$. Thus, by hybrid argument it follows that there exists an integer $0 \leq z < d - 1$ and a distinguisher $\mathcal{D}'$ who can distinguish between the two distributions H^z and H^{z+1} with non-negligible probability ε'.

Let us take a closer look at the hybrids H^z and H^{z+1}: Let g be a gate from layer $z + 1$ with input wires a, b and output wire c.
If the view is taken from H^{z+1} then the garbled table A_g, B_g, C_g, D_g are computed as described in the procedure $\mathcal{P}$, that is, the external values $\Lambda_a, \Lambda_b, \Lambda_c$ are known and thus the key $\mathbf{k}_{c,\Lambda_c}$ is encrypted using keys $\mathbf{k}_{a,\Lambda_a}$ and $\mathbf{k}_{b,\Lambda_b}$ in the

$2\Lambda_a + \Lambda_b$-th entry (the active entry) while the other three (inactive) entries are independent of $\mathbf{k}_{a,\Lambda_a}$, $\mathbf{k}_{b,\Lambda_b}$, $\mathbf{k}_{a,\bar{\Lambda_a}}$ and $\mathbf{k}_{b,\bar{\Lambda_b}}$ (because $\mathcal{P}$ chooses them at random from $(\mathbb{F}_p)^n$).
If the view is taken from H^z then the garbled table of g is computed correctly for all the four entries. Let $\tilde{g}_a$ be a gate whose output wire is a (which is an input wire to gate g); note that by the definitions of the layers $\tilde{g}_a$ must reside at layer no larger than z and thus there is exactly one entry (the active entry) in the garbled table of $\tilde{g}_a$ which encrypts $\mathbf{k}_{a,\bar{\Lambda_a}}$ while the other three (inactive) entries are random values from $(F_p)^n$, therefore reveal no information about $\mathbf{k}_{a,\Lambda_a}$, and more important, no information about $\mathbf{k}_{a,\bar{\Lambda_a}}$. The same observation holds for the gate $\tilde{g}_b$ whose output wire is b. We get that in the computation of the garbled table of gate g (recall that it is in layer $z+1$ and we are currently looking at hybrid H^z) there is exactly one entry (i.e. the active entry) which depends on both $\mathbf{k}_{a,\Lambda_a}$ and $\mathbf{k}_{b,\Lambda_b}$ while the other three (inactive) entries are depend on at least one of $\mathbf{k}_{a,\bar{\Lambda_a}}$ and $\mathbf{k}_{b,\bar{\Lambda_b}}$.

Since the distinguisher $\mathcal{D}'$ has no prior information at all about $\mathbf{k}_{a,\bar{\Lambda_a}}$ and $\mathbf{k}_{b,\bar{\Lambda_b}}$ (i.e. information that achieved from other source but the garbled table of gate g itself in H^z), whenever a computation of F using a key from the vectors $\mathbf{k}_{a,\bar{\Lambda_a}}$ or $\mathbf{k}_{b,\bar{\Lambda_b}}$ is required in order to compute the inactive entries of gate g (in H^z), we could use some other key $\tilde{k}$ instead. Moreover, we could use F without even know $\tilde{k}$ at all, e.g. when working with an oracle.

In the following we exploit the above observation. Let us first define pseudo random function under multiple keys:

Definition 1. *Let $F : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ be an efficient, length preserving, keyed function. F is a* pseudo random function under multiple keys *if for all polynomial time distinguishers $\mathcal{D}$, there is a negligible function neg such that:*

$$|Pr[\mathcal{D}^{F_{\bar{k}}(\cdot)}(1^n) = 1] - Pr[\mathcal{D}^{\bar{f}(\cdot)}(1^n) = 1]| \leq neg(n)$$

where $F_{\bar{k}} = F_{k_1}, \ldots, F_{k_{m(n)}}$ are the pseudo random function F keyed with polynomial number of randomly chosen keys $k_1, \ldots, k_{m(n)}$ and $\bar{f} = f_1, \ldots, f_{m(n)}$ are $m(n)$ random functions $\{0,1\}^n \to \{0,1\}^n$. The probability in both cases is taken over the randomness of $\mathcal{D}$ as well.

It is easy to see (by a hybrid argument) that if F is a pseudo random function then it is a pseudo random function under multiple keys, thus, since the function F used in our protocol is a PRF then for every polynomial time distinguisher $\tilde{\mathcal{D}}$, every positive polynomial p and for all sufficiently large κs:

$$|Pr[\tilde{\mathcal{D}}^{F_{\bar{k}}(\cdot)}(1^\kappa) = 1] - Pr[\tilde{\mathcal{D}}^{\bar{f}(\cdot)}(1^\kappa) = 1]| \leq \frac{1}{p(\kappa)} \quad (3)$$

We are now present a reduction from the indistinguishability between H^z and H^{z+1} to the indistinguishability of the pseudorandom function F under multiple keys. Given the polynomial time distinguisher $\mathcal{D}'$ who distinguishes between H^z and H^{z+1} with non negligible probability ε', we construct a polynomial time distinguisher $\mathcal{D}''$ who distinguishes between F under multiple keys and a set of truly random functions (and thus contradicting the pseudorandomness of F). The distinguisher $\mathcal{D}''$ has an access to $\overline{\mathcal{O}} = \mathcal{O}_1, \ldots, \mathcal{O}_m$ (which is either a PRF under multiple keys or a set of truly random functions), $\mathcal{D}''$ act as follows:

1. Chooses keys and masking values for all players and wires, i.e. $\{k^i_{w,b} \mid w \in W, b \in \{0,1\}, i \in \{1,\ldots,n\}\}$ and $\{\lambda_w \mid w \in W\}$.
2. Constructs the z bottom layers of the circuit C as described in the procedure $\mathcal{P}$, i.e. only the active entry is calculated correctly, the rest three entries are taken to be random from $(\mathbb{F}_p)^n$.
3. Let $w_1, \ldots, w_m$ be the set of wires that enters to the gates in layer $z+1$, such that for every w_i $(1 \le i \le m)$ we have the active signal Λ_{w_i} along with the key $\mathbf{k}_{w_i,\Lambda_{w_i}}$ and the inactive signal $\overline{\Lambda_{w_i}}$ such that the distinguisher has no information about $\mathbf{k}_{w_i,\overline{\Lambda_{w_i}}}$. The distinguisher $\mathcal{D}''$ computes the garbled tables of the gates which belong to layer $z+1$ as described in our protocol with the following exception:
   - Whenever a result of F applied to the key $\mathbf{k}_{w_i,\Lambda_{w_i}}$ is required, it computes it correctly as in our protocol.
   - Whenever a result of F applied to the key $\mathbf{k}_{w_i,\overline{\Lambda_{w_i}}}$ is required, the distinguisher $\mathcal{D}''$ query the oracle $\mathcal{O}_i$ instead.
4. Completes the computation of the garbled circuit, i.e. the garbled tables of the gates which belong to layers $z+2, \ldots, d$, correctly, as in our protocol.
5. Hands the resulting view to $\mathcal{D}'$ and outputs whatever it outputs.

Observe that if $\overline{\mathcal{O}} = F_{\bar{k}}$ then the view that $\mathcal{D}''$ hands to $\mathcal{D}'$ is distributed identically to H^z while if $\overline{\mathcal{O}} = \bar{f}$ then the view that $\mathcal{D}''$ hands to $\mathcal{D}'$ is distributed identically to H^{z+1}. Thus:

$$\begin{aligned}|Pr[\mathcal{D}''^{F_{\bar{k}}(\cdot)}(1^\kappa) = 1] - Pr[\mathcal{D}''^{\bar{f}(\cdot)}(1^\kappa) = 1]| &= \\ |Pr[\mathcal{D}'(H^z) = 1] - Pr[\mathcal{D}'(H^{z+1}) = 1]| &= \varepsilon\end{aligned}$$

where ε is a non-negligible probability (as mentioned above), in contradiction to the pseudo-randomness of F. We conclude that the assumption of the existence of $\mathcal{D}'$ is incorrect and thus:

$$\{\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}\}_{f,\bar{x},k^i_{w,\beta},\lambda j} \overset{c}{\equiv} \{\mathcal{P}(\text{REAL}^{\mathsf{BMR}}_{\mathcal{A}})\}_{f,\bar{x},k^i_{w,\beta},\lambda j}$$

■

C.2 Security in the malicious model

In our protocol there are exactly two points in which a maliciously corrupted party might deviate from the protocol:

- A corrupted party P_c, to whom the circuit input wire w is attached, might cheat in the online phase by sending the external value $\Lambda'_w \neq \lambda_w \oplus \rho_w$, i.e. P_c sends $\overline{\Lambda_w}$.

– A corrupted party might cheat in the offline phase by input a false value as one (or more) of the PRF results of its keys.

It is clear that the first kind of behavior has the same effect as if the adversary inputs to the functionality the value $\bar{\rho_w}$ instead of ρ_w, since $\bar{\Lambda_w} = \lambda_w \oplus \bar{\rho_w}$, and thus, this behavior is permitted to a malicious adversary.

We break the presentation of the security in the malicious case into two steps: first we show that the adversary cannot break the correctness of the protocol with more than negligible probability, and then we use that result (of correctness) in order to show that the joint distributions of the output of the parties in the ideal and real worlds are indistinguishable.

Correctness. Let us denote the event in which a corrupted party cheats by inputting a false PRF result in the offline phase as cheat. In the following we prove the following claim:

Claim 7 *A malicious adversary cannot break the correctness property of our protocol except with a negligible probability. Formally, denote the output of the honest parties in our protocol as Π^J_{SFE} and their output when computed by the functionality f as y_J, for every positive polynomial p and for sufficiently large κ*

$$Pr[\Pi^J_{\mathsf{SFE}} \neq y_J \wedge \Pi^J_{\mathsf{SFE}} \neq \perp \mid \mathsf{cheat}] \leq \frac{1}{p(\kappa)}$$

Proof. To harm the correctness property of the protocol, the adversary has to provide to the offline phase incorrect results of F applied on its keys, such that the resulted garbled circuit will cause the honest party to output some value that is different from y_J. Let GC_{SH} be the garbled circuit resulted by the offline phase in the semi-honest model, i.e. when the adversary provides the correct results of F, and let GC_M be the garbled circuit resulted in the malicious model (such that in both cases the underlying MPC, the adversary and the parties have the same random tape).

Observe that if the adversary succeeds in breaking the correctness then there must be at least one wire c, such that the gate g has input wires a, b and output wire c, and at least one honest party P_j such that in GC_{SH} the active signal that P_j sees is $(v, k^j_{c,v})$ (where $v = \Lambda_c$ is the external value) and in GC_M the active signal is $(\bar{v}, k^j_{c,\bar{v}})$.

In the following analysis we let the adversary more power than it has in reality and assume that it can predict, even before supplying its PRF results (i.e. in the offline phase), which entries are going to be evaluated in the online phase (i.e. it knows the active path). For example, it knows that for some gate g with input wires a, b and output wire c, $\Lambda_a = \Lambda_b = 0$ and thus the active entry for gate g is A_g. In addition, observe that the success probability of the adversary (of breaking the correctness property) is independent for every gate, thus it is sufficient to calculate the success probability of the adversary for a single gate and then multiply the result by the number of gates in the circuit.

So we first analyze the success probability of the adversary to break the correctness of the gate g with input wires a, b and output wire c. Assume, without loss of generality, that the active entry of gate g is A_g which is a vector of n elements from $\mathbb{F}_p$, such that the j-th element of A_g is calculated (as described in functionality 3) by

$$A_g^j = \left(\sum_{i=1}^{n} F_{k_{a,0}^i}(0 \,\|\, j \,\|\, g) + F_{k_{b,0}^i}(0 \,\|\, j \,\|\, g) \right) + k_{c,v}^j \tag{4}$$

For simplicity define

$$\begin{aligned} X^j &\triangleq F_{k_{a,0}^I}(0 \,\|\, j \,\|\, g) + F_{k_{b,0}^I}(0 \,\|\, j \,\|\, g) = \sum_{i \in I} \left(F_{k_{a,0}^i}(0 \,\|\, j \,\|\, g) + F_{k_{b,0}^i}(0 \,\|\, j \,\|\, g) \right) \\ Y^j &\triangleq F_{k_{a,0}^J}(0 \,\|\, j \,\|\, g) + F_{k_{a,0}^J}(0 \,\|\, j \,\|\, g) = \sum_{i \in J} \left(F_{k_{a,0}^i}(0 \,\|\, j \,\|\, g) + F_{k_{b,0}^i}(0 \,\|\, j \,\|\, g) \right) \end{aligned}$$

i.e. X^j is the sum of the PRF results that the adversary provides and Y^j is the sum of the PRF results that the honest player provides. Thus, rewriting equation (4) we obtain

$$A_g^j = X^j + Y^j + k_{c,v}^j$$

In order to break the correctness of gate g, the adversary has to flip the active signal for at least one $j \in J$ (i.e. for at least one honest party), that is, the adversary has to provide false PRF results $\tilde{X}^j$ such that

$$\tilde{A}_g^j = \tilde{X}^j + Y^j + k_{c,\bar{v}}^j$$

Let Δ^j be the difference between the two hidden keys, i.e. $\Delta^j = k_{c,v}^j - k_{c,\bar{v}}^j$, then it follows that $k_{c,\bar{v}}^j = k_{c,v}^j - \Delta^j$ thus in order to make the honest party P_j to evaluate the key $k_{c,\bar{v}}^j$ instead the key $k_{c,v}^j$ the adversary has to set $\tilde{X} = X - \Delta^j$. Then it holds that

$$\begin{aligned} \tilde{X} + Y + k_{c,v}^j &= X - \Delta^j + Y + k_{c,v}^j \\ &= X + Y + k_{c,\bar{v}}^j \\ &= \tilde{A}_g^j \end{aligned}$$

as required and the j-th element (which actually verified by P_j) will be flipped. Observe that in order to succeed the adversary has to find Δ^j. But, since $k_{c,v}^j$ and $k_{c,\bar{v}}^j$ are random element from $\mathbb{F}_p$, the value Δ^j is also a random element from $\mathbb{F}_p$. Note that the adversary provides all the PRF result before the garbled circuit and the garbled inputs are revealed and thus the values that it provides are independent to the garbled circuit (in particular, they are independent of the keys $k_{c,v}^j$ and $k_{c,\bar{v}}^j$). Note that the same analysis holds for the entries B_g, C_g, D_g as well.

Let $\mathsf{flipped\text{-}g}$ be the event in which the adversary succeeds in flipping the signal for at least one honest party P_j in the active entry of gate g, it follows that:

$$\Pr[\mathsf{flipped-g}] = \Pr[\Delta^j = k^j_{c,v} - k^j_{c,\bar{v}}] = \frac{1}{p} < \frac{1}{2^\kappa}$$

Now, assume that when the adversary guesses a wrong Δ^j for some entry of some gate, the parties do not abort and somehow can keep evaluating the circuit using the correct key; then the probability of the adversary to break the correctness of the protocol is just a sum of its success probability for all gates. Let t be a polynomial such that $t(\kappa)$ is an upper bound for the number of gates in the circuit, then by union bound we get:

$$Pr[\Pi^J_{\mathsf{SFE}} \neq y_J \mid \mathsf{cheat}] < \frac{t(\kappa)}{2^\kappa} < \frac{1}{p(\kappa)}$$

for every positive polynomial p. ■

Emulation in the ideal model. In the following we describe the ideal model in which the adversary's view will be emulated, then we show the existence of a simulator $\mathcal{S}'_{\text{OUR}}$ in the malicious model using the simulator $\mathcal{S}_{\text{OUR}}$ in the semi-honest model. The ideal model is as follows:

Inputs. The parties send their inputs ($\bar{x}$) to the trusted party.

Function computed. The trusted party computes $f(\bar{x})$.

Adversary decides. The adversary gets the output y_I and sends to the trusted party whether to 'continue' or 'halt'. If 'continue' then the trusted party sends to the honest parties P_J the output y_J, otherwise then the trusted party sends abort to players P_J.

Outputs. The honest parties output whatever the trusted party sent them while the corrupted parties output nothing. The adversary $\mathcal{A}$ outputs any arbitrary (PPT) function of the initial input of the corrupted parties and the value y_J obtained from the trusted party.

The ideal execution of f on inputs $\bar{x}$, corrupted parties P_I and a security parameter κ is denoted by $\textsc{ideal}^f_{\mathcal{A},I}(1^\kappa, \bar{x})$ and the real execution is denoted by $\textsc{real-mal}^{\mathsf{Our}}_{\mathcal{A},I}(1^\kappa, \bar{x})$; in both cases they refer to the joint distribution of the outputs of all parties.

The reason that the adversary may decide whether the honest parties obtain the output or not is due to the fact that guaranteed output delivery and fairness cannot be achieved with dishonest majority in the general case.

Claim 8 *Our protocol is secure in the malicious model, that is*

$$\textsc{ideal}^f_{\mathcal{A},I}(1^\kappa, \bar{x}) \overset{c}{\equiv} \textsc{real-mal}^{\mathsf{Our}}_{\mathcal{A},I}(1^\kappa, \bar{x})$$

Proof. The simulator $\mathcal{S}'_{\mathrm{OUR}}$ will engage in the ideal computation such that it only gives the input x_I to the trusted party and then receives the output y_I. The simulator $\mathcal{S}'_{\mathrm{OUR}}$ also instruct the trusted party whether to abort or not (i.e. whether to send the honest parties their output). The output of the parties (all of them) in the ideal settings must be indistinguishable to their outputs in the real execution of our protocol.

The idea of the simulation method is that we can use the fact that there exist a simulator $\mathcal{S}_{\mathrm{OUR}}$ in the semi-honest mode, thus, we can construct a garbled circuit that is indistinguishable from one constucted by honest players; and by internally running $\mathcal{A}$ we can extract the exact locations in which $\mathcal{A}$ has cheated.

Before we present the simulator let us define the procedure $\mathcal{P}'$ (Procedure which receives a view simulated by $\mathcal{S}_{\mathrm{OUR}}$ along with a set of keys $\{k^i_{w,j} \mid i \in I, w \in W, j \in \{0,1\}\}$ and rebuilds the garbled circuit just as $\mathcal{P}$ did, but instead of using random keys of its choice it uses the keys received as input for the corrupted parties I.

Procedure 9 (The Procedure $\mathcal{P}'$)

Input. A view v taken from distribution $\textsc{view}^{\mathsf{Our}}_{\mathcal{A}}$ under the input $\bar{x}$; and a set of keys $\mathbf{K_I} = \{k^i_{w,j} \mid i \in I, w \in W, j \in \{0,1\}\}$
Output. A view v' conforming to the message flow in $\textsc{view}^{\mathsf{Our}}_{\mathcal{A}}$ but using the set of keys from the input.
Execute the procedure $\mathcal{P}$ on v with the exception that in step 3a use the keys given as input rather than choosing new ones for every key of parties I.

It is clear that if $\mathbf{K_I}$ given to $\mathcal{P}'$ is chosen randomly then v' and v are indistinguishable, that is let $\mathbf{K_I}$ be the random set of keys, then:

$$\{\textsc{view}^{\mathsf{Our}}_{\mathcal{A}}\}_{\bar{x}} \equiv \{\mathcal{P}'(\textsc{view}^{\mathsf{Our}}_{\mathcal{A}}, \mathbf{K_I})\}_{\bar{x}} \tag{5}$$

In addition we define the procedure $\mathcal{H}$ (Procedure 10) which is given a view $\textsc{view}^{\mathsf{Our}}_{\mathcal{A}}$ and a set of $2n$ PRF results $\mathbf{F_I}$ (computed correctly or not) for the keys of players I. The procedure returns a corresponding view in which the garbled circuit is computed as if it was computed in a real execution of our protocol where the adversary input in the offline phase the PRF results $\mathbf{F_I}$.

Let $\mathbf{K_I}$ as before, be the set of keys generated for the corrupted parties in the offline phase, and $\lambda_{\mathbf{I}}$ be the set of masking values generated for the circuit output wires and for the wires that are attached to the corrupted parties (i.e. the masking values that are in the adversary's view). Note that the PRF results that the corrupted parties input to the functionality (in the offline phase) depends only on the adversary's random tape r, and on the keys and masking values outputted to them from the functionality, that is the PRF results that they provide can be seen as $\mathcal{A}(r, \mathbf{K_I}, \lambda_{\mathbf{I}})$. Since the PRF results that the corrupted parties input to the functionality influence only the resulted garbled gates, in the exact same manner as described in the procedure $\mathcal{H}$, we get the following:

Procedure 10 (The Procedure $\mathcal{H}$)

Input. A view v taken from distribution $\text{VIEW}_{\mathcal{A}}^{\mathsf{Our}}$ under the input $\bar{x}$; and a set of PRF results $\mathbf{F_I}$ of F applied to the set of keys of parties $/I$ (that is, $2n$ PRF results for every key $\{k^i_{w,j} \mid i \in I, w \in W, j \in \{0,1\}\}$
Output. A view v' conforming to the message flow in $\text{VIEW}_{\mathcal{A}}^{\mathsf{Our}}$ but with modified garbled gates according to $\mathbf{F_I}$.
The view v contains all the keys belonging to the corrupted parties I, thus the procedure can tell which of the PRF results in $\mathbf{F_I}$ computed correctly and which is not. Recall that $\mathbf{F_I}$ can be seen as a set of vectors from $(\mathbb{F}_p)^n$, formally, we denote the values in $\mathbf{F_I}$ as:

$$\tilde{F}_{k^i_{w,b}}(0 \parallel 1 \parallel g), \ldots, \tilde{F}_{k^i_{w,b}}(0 \parallel n \parallel g)$$
$$\tilde{F}_{k^i_{w,b}}(1 \parallel 1 \parallel g), \ldots, \tilde{F}_{k^i_{w,b}}(1 \parallel n \parallel g)$$

and the correct PRF values as:

$$F_{k^i_{w,b}}(0 \| 1 \| g), \ldots, F_{k^i_{w,b}}(0 \| n \| g)$$
$$F_{k^i_{w,b}}(1 \| 1 \| g), \ldots, F_{k^i_{w,b}}(1 \| n \| g)$$

for every w, b and $i \in I$.
The procedure changes the garbled gates in the view as follows:
Let g be a gate with input wires a, b and output wire c, from Functionality 3 we can see that

$\tilde{F}_{k^i_{a,0}}(0 \parallel j \parallel g)$ influences A^j_g	$\tilde{F}_{k^i_{b,0}}(0 \parallel j \parallel g)$ influences A^j_g
$\tilde{F}_{k^i_{a,0}}(1 \parallel j \parallel g)$ influences B^j_g	$\tilde{F}_{k^i_{b,1}}(0 \parallel j \parallel g)$ influences B^j_g
$\tilde{F}_{k^i_{a,1}}(0 \parallel j \parallel g)$ influences C^j_g	$\tilde{F}_{k^i_{b,0}}(1 \parallel j \parallel g)$ influences C^j_g
$\tilde{F}_{k^i_{a,1}}(1 \parallel j \parallel g)$ influences D^j_g	$\tilde{F}_{k^i_{b,1}}(1 \parallel j \parallel g)$ influences D^j_g

Thus, for every $\tilde{F}_{k^i_{w,b}}(\alpha \parallel \beta \parallel \gamma)$ of the above, the procedure computes the correct value $F_{k^i_{w,b}}(\alpha \| \beta \| \gamma)$. Then it computes the difference

$$F^{\Delta}_{k^i_{w,b}}(\alpha \parallel \beta \parallel \gamma) = \tilde{F}_{k^i_{w,b}}(\alpha \parallel \beta \parallel \gamma) - F_{k^i_{w,b}}(\alpha \| \beta \| \gamma)$$

Finally, it adds that difference to the appropriate coordinate in one of the vectors A_g, B_g, C_g, D_g as described above. For instance. let $\Delta^i_{a,0} = \tilde{F}_{k^i_{a,0}}(0 \parallel j \parallel g) - F_{k^i_{a,0}}(0 \| j \| g)$ then the procedure adds $\Delta^i_{a,0}$ to the value A_g given in v.
When done with those changes, the procedure output the resulted view v'.

– Let v denote the ensemble $\{\text{VIEW}^{\mathsf{Our}}_{\mathcal{A}}\}_{\bar{x}}$ and $\tilde{v}$ be the same as $\{\text{VIEW}^{\mathsf{Our}}_{\mathcal{A}}\}_{\bar{x}}$ with the exception that the adversary is guaranteed to input correct PRF results to all keys.
– If the set of keys $\mathbf{K}$ and masking values λ in v and $\tilde{v}$ are the same, we get that:

$$v \overset{c}{\equiv} \mathcal{H}(\tilde{v}, \mathcal{A}(r, \mathbf{K_I}, \lambda_\mathbf{I})) \tag{6}$$

We now describe the simulator $\mathcal{S}'_{\text{OUR}}$:

1. The simulator $\mathcal{S}'_{\text{OUR}}$ runs our protocol internally such that it takes the role of the honest parties P_J and the trusted party, and uses the algorithm $\mathcal{A}$ to control the parties P_I. The simulator halt the internal execution right after it receives the external values $\mathbf{\Lambda_I}$ to all the corrupted parties in the online phase. From the internal execution the simulator $\mathcal{S}'_{\text{OUR}}$ can extract the following values:
   (a) Adversary's keys $k^I_{w,0}, k^I_{w,1}$ (in addition to the honest party's keys $k^J_{w,0}, k^J_{w,1}$ since $\mathcal{S}'_{\text{OUR}}$ is the trusted party who chooses them) for every wire w. We denote the set of keys (for both adversary and honest parties) as $\mathbf{K}$.
   (b) Masking values λ for *all* wires, in particular, the masking values of the circuit-input wires that are attached to P_I, i.e. $\lambda_\mathbf{I}$.
   (c) The values $\mathbf{F_I}$, i.e. $2n$ results for every key. Since $\mathcal{S}'_{\text{OUR}}$ is the trusted party in the internal execution, it also knows the PRF results for the honest parties' keys. We denote the set of PRF result (for all keys, both adversary's and honest party's) as $\mathbf{F}$. Moreover, observe that $\mathcal{S}'_{\text{OUR}}$ can check whether $\mathcal{A}$ has cheated in $\mathbf{F_I}$.
   (d) From $\lambda_\mathbf{I}$ and $\mathbf{\Lambda_I}$ the simulator $\mathcal{S}'_{\text{OUR}}$ can conclude $\mathcal{A}$'s input to the functionality x_I.
2. Now focusing on the ideal world, the honest parties and $\mathcal{S}'_{\text{OUR}}$ (this time as the adversary) send their inputs to the trusted party. $\mathcal{S}'_{\text{OUR}}$ sends x_I (that extracted earlier).
3. The simulator $\mathcal{S}'_{\text{OUR}}$ receives the output y_I from the trusted party.
4. $\mathcal{S}'_{\text{OUR}}$ now knows $\mathcal{A}$'s input to the functionality x_I and the output of f on x_I and x_J (where x_J remains hidden to it), it computes $v = \mathcal{S}_{\text{OUR}}(1^\kappa, I, x_I, y_I)$.
5. $\mathcal{S}'_{\text{OUR}}$ computes $v' = \mathcal{P}'(v, \mathbf{K_I})$.
6. $\mathcal{S}'_{\text{OUR}}$ computes $v'' = \mathcal{H}(v', \mathbf{F_I})$ (note that $\mathbf{F_I} = \mathcal{A}(r, \mathbf{K_I}, \lambda_\mathbf{I})$).
7. Having the modified view v'' and the garbled circuit GC_M within it, $\mathcal{S}'_{\text{OUR}}$ now evaluates the circuit on behalf of the honest players. If they abort then $\mathcal{S}'_{\text{OUR}}$ instructs the trusted party to not send the output y_J to P_J (i.e. to output $\perp$). Otherwise, if the evaluation succeeds then $\mathcal{S}'_{\text{OUR}}$ instructs the trusted party to output the correct output y_J. [7]
8. The simulator $\mathcal{S}'_{\text{OUR}}$ outputs the view v'' as the adversary's simulated output.

[7] The decision whether to abort or not is not based on whether the adversary cheated or not, but rather, based on the actual evaluation of the circuit because there might be cases where the adversary cheats and influence only the corrupted parties, e.g. when cheating in i-th PRF values used in a garbled gate of some gate whose output wire is a circuit output wire (where $i \in I$).

Fix the set of keys $\mathbf{K} = \mathbf{K_I} \cup \mathbf{K_J}$ and masking values λ, let $\{\textsc{view}^{\mathsf{Our}}_{\mathcal{A},\mathbf{K},\lambda}\}_{\bar{x}}$ be the probability ensemble of the adversary's view in the real execution of our protocol when the functionality generates the keys $\mathbf{K}$ and masking values λ, $\{\textsc{view}'^{\mathsf{Our}}_{\mathcal{A},\mathbf{K},\lambda}\}_{\bar{x}}$ be the same as $\{\textsc{view}^{\mathsf{Our}}_{\mathcal{A},\mathbf{K},\lambda}\}_{\bar{x}}$ with the exception that here the adversary is guaranteed to input correct PRF results, and let $\{\textsc{view}^{sim}_{\mathcal{A},\mathbf{K},\lambda}\}_{\bar{x}}$ be the probability ensemble of the adversary's view which resulted by $\mathcal{S}'_{\text{OUR}}$ in which the keys and masking values generated by the functionality in the internal execution of the protocol were $\mathbf{K}$ and λ.

From equation 5 it follows that the view v resulted by $\mathcal{S}_{\text{OUR}}$ in step 4 is indistinguishable to the view v' in step 5, that is:

$$\{v\}_{\bar{x}} \equiv \{v'\}_{\bar{x}}$$

also, note that the view v' is exactly the view of the adversary in a real execution of the protocol, in which the adversary provided only correct PRF results, that is

$$\{v'\}_{\bar{x}} \equiv \{v\}_{\bar{x}} \overset{c}{\equiv} \{\textsc{view}'^{\mathsf{Our}}_{\mathcal{A},\mathbf{K},\lambda}\}_{\bar{x}}$$

by equation 6 we get that

$$\begin{aligned}\{\textsc{view}^{sim}_{\mathcal{A},\mathbf{K},\lambda}\}_{\bar{x}} &= \{v''\}_{\bar{x}} \\ &= \{\mathcal{H}(v')\}_{\bar{x}} \\ &\overset{c}{\equiv} \{\mathcal{H}(\textsc{view}'^{\mathsf{Our}}_{\mathcal{A},\mathbf{K},\lambda})\}_{\bar{x}} \\ &\overset{c}{\equiv} \{\textsc{view}^{\mathsf{Our}}_{\mathcal{A},\mathbf{K},\lambda}\}_{\bar{x}}\end{aligned}$$

and since $\mathbf{K}$ and λ are chosen from the same distribution in both cases it follows that the simulated view of the adversary and its view in a real execution of the protocol are the same, that is:

$$\{\textsc{view}^{sim}_{\mathcal{A}}\}_{\bar{x}} \overset{c}{\equiv} \{\textsc{view}^{\mathsf{Our}}_{\mathcal{A}}\}_{\bar{x}}$$

In order to show security in the malicious model the above result is not enough, we also need to show that the joint distribution of the output of all players are indistinguishable. From our previous correctness proof we know that when the honest parties reach the circuit-out wires, they always obtain the correct output (i.e. y_J) except with negligible probability ε . Assume by contradiction the existence of a distinguisher $\mathcal{D}$ and a polynomial q such that

$$|\Pr[\mathcal{D}(\textsc{ideal}^{f}_{\mathcal{S}'_{\text{OUR}},I}(1^\kappa,\bar{x}))=1] - \Pr[\mathcal{D}(\textsc{real-mal}^{\mathsf{Our}}_{\mathcal{A},I}(1^\kappa,\bar{x}))=1]| \geq \frac{1}{q(\kappa)}$$

then we can construct $\mathcal{D}'$ which can distinguish between $\{\textsc{view}^{sim}_{\mathcal{A}}\}_{\bar{x}}$ and $\{\textsc{view}^{\mathsf{Our}}_{\mathcal{A}}\}_{\bar{x}}$ with non negligible probability, $\mathcal{D}'$ is given the view $\mathcal{V}$ and works as follows:

1. Extracts the garbled circuit from the view $\mathcal{V}$, and evaluates it on behalf of the honest parties P_J to obtain y_J.
2. Hands $\{y_J, \mathcal{V}\}$ to D and output whatever it outputs.

if $\mathcal{V}$ is the adversary's view in the real execution of the protocol then

$$\{y_J, \mathcal{V}\} \equiv \textsc{real-mal}^{\mathsf{Our}}_{\mathcal{A},I}(1^\kappa, \bar{x})$$

otherwise, if $\mathcal{V}$ is the output of $\mathcal{S}'_{\text{OUR}}$ then

$$\{y_J, \mathcal{V}\} \equiv \textsc{ideal}^{f}_{\mathcal{S}'_{\text{OUR}},I}(1^\kappa, \bar{x})$$

thus:

$$\begin{aligned}\Pr[\mathcal{D}'(\{\textsc{view}^{\mathsf{Our}}_{\mathcal{A}}\}_{\bar{x}}) = 1] &= \Pr[\mathcal{D}(\textsc{real-mal}^{\mathsf{Our}}_{\mathcal{A},I}(1^\kappa, \bar{x})) = 1]\\ \Pr[\mathcal{D}'(\{\textsc{view}^{sim}_{\mathcal{A}}\}_{\bar{x}}) = 1] &= \Pr[\mathcal{D}(\textsc{ideal}^{f}_{\mathcal{S}'_{\text{OUR}},I}(1^\kappa, \bar{x})) = 1]\end{aligned}$$

and so

$$|\Pr[\mathcal{D}'(\{\textsc{view}^{\mathsf{Our}}_{\mathcal{A}}\}_{\bar{x}}) = 1] - \Pr[\mathcal{D}'(\{\textsc{view}^{sim}_{\mathcal{A}}\}_{\bar{x}}) = 1]| \geq \frac{1}{q(\kappa)}$$

which is a contradiction to the above result. Thus we conclude that

$$\textsc{ideal}^{f}_{\mathcal{S}'_{\text{OUR}},I}(1^\kappa, \bar{x}) \overset{c}{\equiv} \textsc{real-mal}^{\mathsf{Our}}_{\mathcal{A},I}(1^\kappa, \bar{x})$$

■