J Cryptol (2019) 32:1026–1069
https://doi.org/10.1007/s00145-019-09322-2

Journal of
CRYPTOLOGY

Efficient Constant-Round Multi-party Computation Combining BMR and SPDZ*

Yehuda Lindell · Benny Pinkas
Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
benny@pinkas.net

Nigel P. Smart
Department of Computer Science, University of Bristol, Bristol, UK
imec-COSIC, KU Leuven, Leuven, Belgium

Avishay Yanai
Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel

Communicated by Jonathan Katz.

Received 29 February 2016 / Revised 4 April 2019
Online publication 26 April 2019

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 a *constant-round* multi-party secure computation 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 protocol of Beaver et al. (the BMR protocol) and is the first version of that protocol that is *concretely* efficient for the dishonest majority case. 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.

Keywords. Secure multiparty computation (MPC), Garbled circuits, Concrete efficiency, BMR, SPDZ.

*An extended abstract of this paper appeared at *CRYPTO 2015*; this is the full version.

1. Introduction

1.1. *Background and Prior Work*

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 [14,32], even in the presence of malicious adversaries. When an honest majority (or a 2/3 majority) is assumed, security can even be obtained information theoretically [5,6,30]. In this paper, we focus on the problem of obtaining 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 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 [12,22–24,26] for just a few examples. In general, there are two approaches that have been followed: The first uses Yao's garbled circuits [32] and the second utilizes interaction for every gate like the GMW protocol [14].

There are extremely efficient variants of Yao's protocol for the two-party case that are secure against malicious adversaries (e.g., [23,24]). These protocols run in a constant number of rounds and therefore remain fast over high latency networks. The BMR protocol [1,31] 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, the original BMR protocol suffers from two main drawbacks:

- The protocol uses circuit-based zero-knowledge proofs to ensure that the parties input correct values obtained from the pseudorandom generator in the protocol. This requires a very large proof (of a circuit computing a pseudorandom generator) for every gate of the circuit. Thus, the protocol serves as a feasibility result for achieving constant-round MPC, but cannot be run in practice.
- The original BMR protocol only guarantees security for malicious adversaries if at most a *minority* of the parties are corrupt. This is due to the fact that constant-round protocols for multi-party commitment, coin tossing, and zero knowledge were not known at the time for the setting of dishonest majority. The existence of constant-

round protocols for multi-party secure computation in the presence of a dishonest majority was proven later in [18,27]. However, these too are feasibility results and are not concretely efficient.

The TinyOT and SPDZ protocols [12,26] follow the GMW paradigm and have separate offline and online phases. Both of these protocols overcome the issues of the BMR protocols 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 for the case of an honest majority appears in [9] and for the case of a dishonest majority in [17]; however, their construction appears to be "concretely inefficient." In a setting of 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 each level of the circuit. This results in a large amount of interaction and high latency, especially when the parties wish to compute *deep* circuits over slow networks (e.g., the Internet). To sum up, prior to this work, there was no known concretely efficient *constant-round* protocol for the multi-party and dishonest majority setting (with the exception of [7] that considers the specific three-party case only). We therefore address this setting.

1.2. *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 and a dishonest majority. Our protocol has 12 communication rounds, of which only 3 rounds are in the online phase. This makes it much more efficient than prior protocols [12,26] for deep circuits and/or slow networks, since prior works all have a number of rounds that is (at least) the depth of the circuit being computed. The basic idea behind the construction is to use an efficient (either constant or 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). Our main conceptual contribution, resulting in a great performance improvement, is to show that it is not necessary for the parties to prove (expensive) zero-knowledge proofs that they used the correct pseudorandom generator values in the offline circuit generation phase. 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 MPC protocol (either constant or non-constant round), we provide an optimized version that utilizes specific features of the SPDZ protocol [12].

In our general construction, the new constant-round 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 a 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[1] 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 this only causes the honest parties to abort (which is inevitable in the dishonest majority anyway). 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 evaluate compute the garbled circuit. The online phase is therefore very fast.

In our concrete instantiation of the protocol using SPDZ [12], 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 on 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 inputs and evaluate the function; this phase just involves exchanging shares of the circuit and garbled values on the input wires and locally evaluate 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 a 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 with 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.

Subsequent Work. Following our work, there has been renewed interest in the BMR protocol. First and foremost, the works of [16,25] build directly on this work and show how to construct the BMR garbled circuit more efficiently. In addition, [3] considered the semi-honest setting and [21] apply an optimized version of our construction to efficient RAM-based MPC. The fact that constant-round BMR protocol outperform secret-sharing-based protocols for not-shallow circuits and slow (Internet-like) networks has been demonstrated in [3,4,21].

[1] In our construction, we use a pseudorandom function as opposed to a pseudorandom generator used in the original BMR [1].

1.3. *Paper Structure*

In Sect. 2, we give a detailed description of the BMR protocol. In Sect. 3, we present our general protocol that can utilize any MPC protocol for arithmetic circuits as a subprocedure. Then, in Sect. 4, we describe our specific BMR protocol that uses SPDZ [12] as the underlying MPC protocol, and we analyze its complexity. We utilize specific properties of SPDZ for further optimizations, in order to obtain an even more efficient evaluation. Finally, we provide a full proof of our construction in Sect. 5.

2. Background—The BMR Protocol [1]

We outline the basis of our protocol, which is the protocol of Beaver, Micali, and Rogaway against a semi-honest adversary.[2] The protocol is comprised of an offline phase and an online phase. In the offline phase, the garbled circuit is constructed by the parties, while in the online phase, a matching set of garbled inputs is exchanged between the parties and each party evaluates it locally.

We now describe the main elements of the BMR protocol. Let κ denote the computational security parameter, n denote the number of parties, and let $[n] = \{1, \ldots, n\}$. The wires in the circuit that computes the function f are indexed $0, \ldots, W-1$. The protocol is based on the following key components:

Seeds and Superseeds. Two random seeds are associated with each wire in the circuit by each party. We denote the 0-seed and 1-seed that are chosen by party P_i ($i \in [n]$) for wire w as $s^i_{w,0}$ and $s^i_{w,1}$ such that $s^i_{w,j} \in \{0,1\}^\kappa$. During the garbling process, the parties 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 parties, 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. Denote $L = |S_{w,j}| = n \cdot \kappa$.

Garbling Wire Values. For each gate g that 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 $M \bigoplus_{i=1}^n G(s^i_{w,0})$, where G is a pseudorandom 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 each wire w independently. These masking bits remain unknown to the parties during the entire protocol, thereby preventing them from knowing the *real values* ρ_w that actually pass through the wires. The values that the parties *do* know are called the *external values*, denoted Λ_w. An external value is defined to be the exclusive-or of the real value and the masking value; that is, $\Lambda_w = \rho_w \oplus \lambda_w$. When evaluating the garbled circuit, the parties

[2] Their original work also offers a version against a malicious adversary; however, it requires an honest majority and is not concretely efficient.

only see the external values of the wires, which are random bits that reveal 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; hence, 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 inputs $s^i_{a,0}, s^i_{a,1}, s^i_{b,0}, s^i_{b,1}, s^i_{c,0}, s^i_{c,1}$. Thus, the appropriate superseeds 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 $S_{\alpha,\beta} = s^1_{\alpha,\beta} \| \cdots \| s^n_{\alpha,\beta}$. In addition, P_i also inputs the output of a pseudorandom generator G applied to each of its seeds, and its share of the masking bits, i.e., $\lambda^i_a, \lambda^i_b, \lambda^i_c$ (where $\lambda_w = \bigoplus_{i=1}^n \lambda^i_w$).

The output is the garbled gate of g which is 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 wire a and one superseed for wire b it is possible 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 1.

The BMR Online Phase. In the online phase, the parties only have to obtain one superseed for every circuit-input wire, and then every party can evaluate the circuit on its own, without interaction with the rest of the parties. Formally, Protocol 1 realizes the online phase.

Functionality 1. (garble-gate-BMR)

Let κ denote the security parameter, and let $G : \{0,1\}^\kappa \to \{0,1\}^{2n\kappa}$ be a pseudorandom 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

- **Seeds:** Party P_i inputs uniform $s^i_{x,b} \in \{0,1\}^\kappa$ for $x \in \{a,b,c\}$ and $b \in \{0,1\}$.
- **Stretched seeds:** Party P_i inputs the L-bit strings $\tilde{\Upsilon}^i_{x,b}$ and $\Upsilon^i_{x,b}$ for $x \in \{a,b,c\}$ and $b \in \{0,1\}$. (If P_i is honest, then $\tilde{\Upsilon}^i_{x,b} = G^1(s^i_{x,b})$ and $\Upsilon^i_{x,b} = G^2(s^i_{x,b})$; that is, $\tilde{\Upsilon}^i_{x,b}$ and $\Upsilon^i_{x,b}$ are the outputs of the pseudorandom generator G on the seed $s^i_{x,b}$).
- **Masking bits.** Party P_i inputs a uniform $\lambda^i_x \in \{0,1\}$ for $x \in \{a,b,c\}$.

Outputs: The functionality first computes $\lambda_x = \bigoplus_{i=1}^n \lambda^i_x$ for $x \in \{a,b,c\}$. 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):

$$A_g = \tilde{\Upsilon}^1_{a,0} \oplus \cdots \oplus \tilde{\Upsilon}^n_{a,0} \oplus \tilde{\Upsilon}^1_{b,0} \oplus \cdots \oplus \tilde{\Upsilon}^n_{b,0} \oplus \begin{cases} S_{c,0} & \text{if } f_g(\lambda_a, \lambda_b) = \lambda_c \\ S_{c,1} & \text{otherwise} \end{cases}$$

$$B_g = \Upsilon^1_{a,0} \oplus \cdots \oplus \Upsilon^n_{a,0} \oplus \tilde{\Upsilon}^1_{b,1} \oplus \cdots \oplus \tilde{\Upsilon}^n_{b,1} \oplus \begin{cases} S_{c,0} & \text{if } f_g(\lambda_a, \bar{\lambda_b}) = \lambda_c \\ S_{c,1} & \text{otherwise} \end{cases}$$

$$C_g = \tilde{\Upsilon}^1_{a,1} \oplus \cdots \oplus \tilde{\Upsilon}^n_{a,1} \oplus \Upsilon^1_{b,0} \oplus \cdots \oplus \Upsilon^n_{b,0} \oplus \begin{cases} S_{c,0} & \text{if } f_g(\bar{\lambda_a}, \lambda_b) = \lambda_c \\ S_{c,1} & \text{otherwise} \end{cases}$$

$$D_g = \Upsilon^1_{a,1} \oplus \cdots \oplus \Upsilon^n_{a,1} \oplus \Upsilon^1_{b,1}) \oplus \cdots \oplus \Upsilon^n_{b,1} \oplus \begin{cases} S_{c,0} & \text{if } f_g(\bar{\lambda_a}, \bar{\lambda_b}) = \lambda_c \\ S_{c,1} & \text{otherwise} \end{cases}$$

Protocol 1. (Protocol BMR-online phase)

Step 1—send values:

1. Every party P_i broadcasts the external values on the wires associated with its input. At the end of this step, the parties 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 party P_i broadcasts one seed for each circuit-input wire, namely, the Λ_w-seed. At the end of this step, the parties know the Λ_w-superseed for every circuit-input wire.

Step 1—evaluate circuit: The parties 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 of that gate (see [1] 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.

Broadcast. As described in Protocol 1, in the online phase the parties are instructed to broadcast one key per input wire. In the semi-honest setting, broadcasting a value simply takes one communication round in which the sender sends its value to all other parties.[3] However, in the malicious setting a corrupted sender may send the value v to one party and a different value $v' \neq v$ to another party. In the setting of $t < n/3$ *corrupted parties*, fully secure broadcast (without abort) can be achieved in an expected constant number of rounds [13] (or deterministically in t rounds [28]). In the setting of $t \geq n/3$ *corrupted parties, and in particular with no honest majority at all*, fully secure broadcast (without abort) can only be achieved using a public key infrastructure and with t rounds of communication. However, we do *not* actually need a fully secure broadcast, since we allow an adversary to cause some parties to abort. Thus, we can use a simple two-round echo-broadcast protocol; this has been shown to be sufficient for secure computation with abort [15] (and even UC secure). In more detail, the echo-broadcast works by having the sender send its message v to all parties $P_1, \ldots, P_n$ in the first round, and then every party P_i sends (echoes) the value that it received in the first round to all other parties. If any party received two different values, then it aborts. Otherwise, it outputs the unique value that it saw. It is not difficult to show that if the dealer is honest, then all honest parties either output the dealer's message v or abort (but no other value can be output). Furthermore, if the dealer is dishonest, then there is a unique value v such that every honest party either outputs v or aborts. See [15] for more details. We therefore write our protocol assuming a broadcast channel and utilize

[3] We assume that the parties interact over a point-to-point network.

the above protocol to achieve broadcast in the point-to-point setting. As a result, our protocol has a constant number of rounds even in the point-to-point model.

3. The General Protocol

3.1. *A 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 above. First, instead of using the 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 [5] 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*. Consequently, we naturally replace the seeds $s^i_{x,b}$, which are bit strings input to a pseudorandom generator, that the parties input with keys $k^i_{x,b}$, which are field elements input to a pseudorandom function.

Second, we observe that the external values of wires[4] 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 checked inside the secure computation. This would mean 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 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 values from $\{0,1\}^\kappa$ (for each vector coordinate), rather than a much larger finite field that can encode all of $\{0,1\}^L$. Due to this, the parties choose *keys* (for a pseudorandom function) rather than *seeds* for a pseudorandom generator. The keys that

[4] 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.

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 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 only

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

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

Functionality 2. (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 $(input, P_i, x_i)$ from P_i and $(input, P_i, ?)$ from all other parties, the functionality stores (P_i, x_i). Upon having (P_i, x_i) for all $i \in [n]$ the functionality 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.

The goal of this paper is to present a protocol Π_{SFE} that realizes Functionality 2 in a constant number of rounds in the setting of a malicious adversary who may corrupt up to $n - 1$ parties. 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 3. The relation diagram between functionalities and protocols presented in this paper is presented in Fig. 1. The generic MPC functionality $\mathcal{F}_{\mathsf{MPC}}$ is *reactive*. We require a *reactive* MPC functionality because our protocol Π_{SFE}

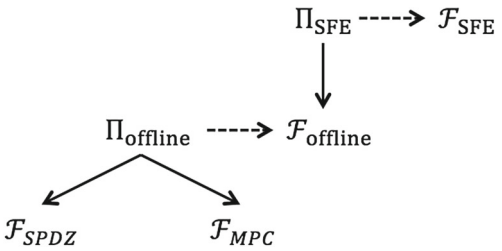

Fig. 1. Outline of our construction: Dashed lines mean that Π securely realizes $\mathcal{F}$; solid lines mean that Π is constructed in the $\mathcal{F}$-hybrid model.

will make repeated sequences of calls to $\mathcal{F}_{\text{MPC}}$ 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.

In what follows, we assume that the $\mathcal{F}_{\text{MPC}}$ functionality maintains a data structure in which it stores its internal values, so that the parties may request to perform operations (i.e., **Input, Output, Add, Multiply**) over the entries of the data structure. We use the notation $[val]$ to represent the key used by the functionality to store value val. In addition, we use the arithmetic shorthands $[z] = [x] + [y]$ and $[z] = [x] \cdot [y]$ to represent the result of calling the **Add** and **Multiply** commands over the inputs x, y and output z. That is, after calling $[z] = [x] + [y]$ (resp. $[z] = [x] \cdot [y]$) the key $[z]$ is associated with the value $x + y$ (resp. $x \cdot y$).

In the **Output** command of $\mathcal{F}_{\text{MPC}}$ (Functionality 3), $i = 0$ means that the value indexed by $varid$ is output to all parties and $i \neq 0$ means that it is output to party P_i only. In both cases, the adversary has the power to decide if an honest party receives the output value or not (where in the latter case, it aborts). Furthermore, when $i = 0$, the adversary has the ability to inspect that value before deciding whether to abort.[5]

[5]Recall that we write our protocol assuming a broadcast channel. Thus, even though we write that in the output stage all parties receive output if $i = 0$, when instantiating the broadcast channel with the simple echo-broadcast described in Sect. 2, some of the honest parties may receive the output and some may abort.

Functionality 3. (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 (otherwise, i.e., if the parties do not agree on p, the functionality halts). 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 $i \neq 0$.

3.2. *The Offline Functionality:* preprocessing-I *and* preprocessing-II

Our protocol, Π_{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 4. In Sect. 4, 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 A" we present a generic implementation of $\mathcal{F}_{\mathsf{offline}}$ based on any underlying protocol Π_{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 phase 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, the preprocessing-II phase 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 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. 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.

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

In Protocol 2, we present our protocol Π_{SFE} for securely computing $\mathcal{F}_{\mathsf{SFE}}$ in the $\mathcal{F}_{\mathsf{offline}}$-hybrid model. In this paper, we prove the following:

Theorem 1. (Main Theorem) *If F is a pseudorandom function, then Protocol Π_{SFE} securely computes $\mathcal{F}_{\mathsf{SFE}}$ in the $\mathcal{F}_{\mathsf{offline}}$-hybrid model, in the presence of a static malicious adversary corrupting any number of parties.*

Functionality 4. (The Offline Functionality—$\mathcal{F}_{\mathsf{offline}}$ (Part 1))

This functionality runs the same **Initialize**, **Wait**, **InputData**, and **Output** commands as $\mathcal{F}_{\mathsf{MPC}}$ (Functionality 3). In addition, the functionality has two additional commands **preprocessing-I** and **preprocessing-II**, as follows.

preprocessing-I: On input (preprocessing-I, W, G), for every wire $w \in [1, \ldots, W]$:

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

Part 2 in Figure 5.

Functionality 5. (The Offline Functionality—$\mathcal{F}_{\mathsf{offline}}$ (Part 2))

preprocessing-II: On input (preprocessing-II, C_f) for a Boolean circuit C_f with up to W wires and G gates.

- For all wires w that are attached to party P_i open λ_w to party P_i by running **Output** as in functionality $\mathcal{F}_{\mathsf{MPC}}$.
- For all output wires w open λ_w to all parties by running **Output** as in functionality $\mathcal{F}_{\mathsf{MPC}}$.
- For every gate g with input wires a, b and output wire c (with $0 \leq a, b, c < W$):
  - Party P_i provides the following values for $x \in \{a, b\}$ by running **InputData** as in functionality $\mathcal{F}_{\mathsf{MPC}}$:

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

    (Note that the functionality just receives these values from the parties and does not check that these are generated from F in any specific way. Nevertheless, for the sake of clarity, we chose not to introduce more variables here and we use the notation $F()$ as the parties' inputs.)
  - Define the selector variables

$$\chi_1 = \begin{cases} 0 & \text{if } f_g(\lambda_a, \lambda_b) = \lambda_c \\ 1 & otherwise \end{cases} \qquad \chi_2 = \begin{cases} 0 & \text{if } f_g(\lambda_a, \overline{\lambda_b}) = \lambda_c \\ 1 & otherwise \end{cases}$$
$$\chi_3 = \begin{cases} 0 & \text{if } f_g(\overline{\lambda_a}, \lambda_b) = \lambda_c \\ 1 & otherwise \end{cases} \qquad \chi_4 = \begin{cases} 0 & \text{if } f_g(\overline{\lambda}_a, \overline{\lambda_b}) = \lambda_c \\ 1 & otherwise \end{cases}$$

  - 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 $1 \leq j \leq n$:

$$A^j_g = \left(\sum_{i=1}^{n}(F_{k^i_{a,0}}(0\|j\|g) + F_{k^i_{b,0}}(0\|j\|g))\right) + k^j_{c,\chi_1}$$
$$B^j_g = \left(\sum_{i=1}^{n}(F_{k^i_{a,0}}(1\|j\|g) + F_{k^i_{b,1}}(0\|j\|g))\right) + k^j_{c,\chi_2}$$
$$C^j_g = \left(\sum_{i=1}^{n}(F_{k^i_{a,1}}(0\|j\|g) + F_{k^i_{b,0}}(1\|j\|g))\right) + k^j_{c,\chi_3}$$
$$D^j_g = \left(\sum_{i=1}^{n}(F_{k^i_{a,1}}(1\|j\|g) + F_{k^i_{b,1}}(1\|j\|g))\right) + k^j_{c,\chi_4}$$

  - Store values $[\mathbf{A}_g]$, $[\mathbf{B}_g]$, $[\mathbf{C}_g]$, $[\mathbf{D}_g]$.

Protocol 2. (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.

Preprocessing: 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 gate's 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.

3.4. *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 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 might input inconsistent values. For example, the masking value λ_w that is common to a number of gates (which happens when some wire enters more than one gate) needs to be identical in all of these gates. In addition, the pseudorandom function values might 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 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 the potential attack against consistency 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. This introduces a challenge since the functionality $\mathcal{F}_{\mathsf{MPC}}$ works (i.e., its commands are) over a finite field $\mathbb{F}_p$ while the masking bit must be a single binary bit. Thus, it is not possible to simply have each party choose its own share bit and then XOR these bits inside $\mathcal{F}_{\mathsf{MPC}}$. The only option offered by $\mathcal{F}_{\mathsf{MPC}}$ is to have the parties each input a field element, and then use these inputs to produce a uniform bit that will be used to mask the wire's signal. This must be done in a way that results in a uniform value in $\{0, 1\}$, even in the presence of malicious parties may input field elements that are not according to the prescribed distribution, and potentially harm the security. This must therefore be prevented by our protocol.

In "Appendix A," we describe a *general* method for securely computing $\mathcal{F}_{\mathsf{offline}}$ in the $\mathcal{F}_{\mathsf{MPC}}$-hybrid model, using *any* protocol that securely computes the $\mathcal{F}_{\mathsf{MPC}}$ ideal functionality. The aforementioned problem 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 $\lambda_w = \prod_{i=1}^n \lambda^i_w \notin \{-1, +1\}$, then this will be discovered since the circuit outputs a nonzero value for

$(\prod_{i=1}^{n} \lambda_w^i)^2 - 1$, 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 show that *all* of these complications can be removed 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.

4. The SPDZ-Based Instantiation

4.1. *Utilizing the SPDZ Protocol*

As discussed in Sect. 3.1, 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 protocol as the underlying implementation of $\mathcal{F}_{\mathsf{MPC}}$. These optimizations are possible because the SPDZ protocol provides a richer interface to the protocol designer than the naive generic MPC interface given in the 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 Sect. 4.2, 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 Sect. 4.3, and the expected results from an implementation of the circuit using the SPDZ protocol are in Sect. 4.6.

Throughout, we utilize $\mathcal{F}_{\mathsf{SPDZ}}$ (Functionality 6), which represents an idealized representation of the SPDZ protocol, akin to the functionality $\mathcal{F}_{\mathsf{MPC}}$ from Sect. 3.1. 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 multiplication (Beaver) Triples[6] for use in the multiplication gates performed in the SPDZ online phase [11]. 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 (SHE) to produce the shared random values/bits and the Beaver multiplication triples, is given in [11].

[6] Multiplication (Beaver) triples are a standard part of the implementation of the SPDZ protocol; we assume familiarity with this method in this paper.

4.2. *The* Π_{offline} *SPDZ-Based Protocol*

As remarked earlier, $\mathcal{F}_{\mathsf{offline}}$ can be securely computed using *any* secure multi-party 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, this option 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, 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.

Functionality 6. (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. The functionality will accept a maximum of M **Multiply**, B **RandomBit**, R **Random** commands overall in addition to I **InputData** commands per party. If the number of command requests exceeds the above, then the functionality aborts.

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 $i \neq 0$.

preprocessing-I.

1. **Initialize the MPC Engine**[7]: Call **Initialize** on the functionality $\mathcal{F}_{\mathsf{SPDZ}}$ with input p, a prime with $p > 2^k$ and with parameters

$$M = G_X(2n + 3) + G_A(4n + 5), \quad B = W,$$
$$R = 2 \cdot W \cdot n, \quad I = 8 \cdot G \cdot n,$$

[7] By "MPC Engine," we refer to the underlying secure computation, the SPDZ functionality in this case.

where G_X, G_A are the number of XOR and AND gates in C_f, respectively, 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. The value of M is derived from the complexity analysis below and $I = 8 \cdot G \cdot n$ since every gate has 2 input wires, each input wire has 2 keys per party, who inputs 2 pseudorandom function outputs values per party.

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 $[k^i_{w,j}]$. The parties then call **Output** to open $[k^i_{w,j}]$ to party i for all j and w. The vector of shares $[k^i_{w,j}]^n_{i=1}$ shall be denoted by $[\mathbf{k}_{w,j}]$.

preprocessing-II. (This protocol implements the computation of gate tables as it is detailed in the BMR protocol. The correctness of this construction is explained at the end of "Sect. 2.")

1. **Output input wire values:** For all wires w which are attached to party P_i (i.e., correspond to input bits of 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 $\mathbf{k}_{c,0} = k^1_{c,0}, \ldots, k^n_{c,0}$ or $\mathbf{k}_{c,1} = k^1_{c,1}, \ldots, k^n_{c,1}$; which vector is hidden 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 differently, as follows (concrete examples are given after the preprocessing specification):

$$[x_a] = \left(f_g([\lambda_a], [\lambda_b]) \stackrel{?}{\neq} [\lambda_c] \right) = (f_g([\lambda_a], [\lambda_b]) - [\lambda_c])^2$$

$$[x_b] = \left(f_g([\lambda_a], [\overline{\lambda_b}]) \stackrel{?}{\neq} [\lambda_c] \right) = (f_g([\lambda_a], (1 - [\lambda_b])) - [\lambda_c])^2$$

$$[x_c] = \left(f_g([\overline{\lambda_a}], [\lambda_b]) \stackrel{?}{\neq} [\lambda_c] \right) = (f_g((1 - [\lambda_a]), [\lambda_b]) - [\lambda_c])^2$$
$$[x_d] = \left(f_g([\overline{\lambda_a}], [\overline{\lambda_b}]) \stackrel{?}{\neq} [\lambda_c] \right) = (f_g((1 - [\lambda_a]), (1 - [\lambda_b])) - [\lambda_c])^2$$

where the binary operator $\stackrel{?}{\neq}$ is defined as $[a] \stackrel{?}{\neq} [b]$ 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 calculations 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:

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

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 calculations 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 these values by calling **InputData** on the functionality $\mathcal{F}_{\mathsf{SPDZ}}$. The resulting input pseudorandom vectors are denoted by

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

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

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

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.

4.3. *Circuit Complexity*

In this section, we analyze the complexity of the circuit that constructs the garbled version of C_f in terms of the number of multiplication gates and the depth of the circuit.[8] We are mainly 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 to have a negligible overhead. We are interested in the depth of the circuit because it gives a lower bound on the number of rounds of interaction that are required for computing the circuit. (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 in the same order as it appears in Step 3 of preprocessing-II phase:
  1. In order to calculate the indicators in Step 3a, it suffices to compute one multiplication and four squarings. 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 $[x_d] = (1 - [\lambda_a] - [\lambda_b] + [t] - [\lambda_c])^2$.

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

[8] This analysis refers to the complexity of the circuit that the parties garble in the offline phase, not the circuit that the parties wish to compute over their private inputs (i.e., C_f).

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 $4n$ multiplications (since we multiply the bit $[x_a]$ with each component of the vector $([\mathbf{k}_{c,1}] - [\mathbf{k}_{c,0}])$. The same holds for bits $[x_b]$, $[x_c]$, and $[x_d]$). Note that in XOR gates only $2n$ 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 (and counting a squaring operation as a multiplication), we have $4n+5$ multiplications per AND gate and $2n + 3$ multiplications per XOR gate.

- **Multiplications in the entire circuit.** Denote the number of multiplication operations per gate (i.e., $4n + 5$ for AND and $2n + 3$ for XOR) by c. We get $G \cdot c$ multiplications for garbling all gates (where G is the number of gates in C_f). Besides garbling the gates, we have no other multiplication operations. 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. 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 preprocessing 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 $\mathcal{F}_{\mathsf{SPDZ}}$.[9] 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 $2 \cdot n \cdot W$.

4.4. *Communication and Computation Complexity*

Denote by W_I and W_O the number of input and output wires in C_f. We first analyze the communication complexity of our online phase and then the offline. We count the number of underlying operations in $\mathcal{F}_{\mathsf{MPC}}$ and then plug in the complexity of these

[9] 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.

operations when using SPDZ [12]. In addition, we count the number of bit/element broadcasts, which will be replaced later with $O(n^2)$ using the simple broadcast with abort protocol discussed above.

Online Phase. The parties first broadcast the external bit for w, which is used to open the appropriate key for w for every $w \in W_I$. Then, the parties open the garbled version of C_f (i.e., the four entries $\mathbf{A}_g, \mathbf{B}_g, \mathbf{C}_g, \mathbf{D}_g$) by calling **Output** $4Gn$ times, where G is the number of gates in the circuit. Overall, W_I bits are broadcast and $W_I + 4Gn$ field elements are opened. The **Output** command in SPDZ has communication and computation complexity of $O(n^3)$; plugging this into the above, we obtain communication complexity $O(W_I \cdot n^3 + G \cdot n^4)$ and computation complexity $O(G \cdot n^4)$. (Note that the bit broadcast operations require essentially no computation.)

Offline Phase. Our offline phase consists of two subphases, preprocessing-I and preprocessing-II, which are SPDZ's offline and online phases, respectively. In the former, the parties generate the raw materials like multiplication triples and input pairs (see [12]), while in the latter they evaluate the arithmetic circuit that produces the garbled circuit of C_f. We count the complexity of preprocessing-II first: We count the number of command invocations and then plug in SPDZ's complexities. For all $w \in W_I$, the protocol runs **Output** for the masking bit of input wire w to the party with whom it is associated, and for all $w \in W_O$ the protocols runs **Output** of the masking bit of w to all parties. Then, as mentioned before, the parties input $I = 8Gn$ PRF outputs for the computation of the garbled circuit using the **Input** command. Finally, there are $O(n)$ invocations of **Multiply** to garble each gate (specifically, $4n + 5$ for an AND gate and $3n + 2$ for a XOR gate). Performing both **Input** and **Multiply** in SPDZ takes $O(n)$ communication and computation, and thus we get overall $O\left((W_I + W_O) \cdot n^3 + G \cdot n^2\right)$ communication and computation.

As for preprocessing-I, we take the complexity analysis from [19]. Generating an input pair requires the communication of $(n-1) \cdot (\kappa^2 + \kappa)$ bit, resulting in $O(n^2 \cdot \kappa^2 \cdot G)$ for all inputs. Generating a multiplication triple costs $O(n^2 \cdot \kappa^2)$ and we need $O(n)$ multiplication triples per gate, resulting in $O(G \cdot n^3 \cdot \kappa^2)$ bits of communication.

4.5. *Round Complexity*

As analyzed above, the circuit that constructs the garbled version of C_f has multiplication depth of three. It therefore remains to plug in the round complexity of the SPDZ offline phase and its implementation of the commands in $\mathcal{F}_{\mathsf{MPC}}$. This yields an overall complexity of 12 rounds of communication: 6 rounds for preprocessing-I (SPDZ offline), 3 rounds for preprocessing-II (SPDZ online), and 3 rounds for the online phase to evaluate the garbled circuit.

Table 1. SPDZ offline generation times in milliseconds per operation.

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

4.6. *Expected Runtimes*

To estimate the run time of our protocol, we extrapolate from known public data [11,12]. (This involves some speculation, but is based on real values for actual cost of the SPDZ operations, which dominates the computation and communication.) The offline phase of the protocol runs both the offline and online phases of the SPDZ protocol. The numbers that are listed in Table 1 refer (in milliseconds) to the SPDZ offline phase, as described in [11], with covert security and a 20% probability of cheating, using finite fields of size 128 bits. As described in [11], comparable times are obtainable for running in the fully malicious mode (but more memory is needed). The offline phase of SPDZ is comprised of the generation of several types of raw material: The multiplication (Beaver) triples are used by the parties to perform a secure multiplication over two variables stored by the functionality; the number of required multiplication triples is equivalent to the number of multiplication gates in the arithmetic circuit that we use to construct the garbled circuit C_f. The number of random bits (resp. random field elements) is the number of bits (resp. field elements) that will be used in the arithmetic circuit. Finally, we also consider the number of required inputs since the SPDZ's offline phase produces some raw data for every input wire in the circuit; these raw data are essentially an authenticated share of a random element $[r]$ which is opened only to the party to which that input wire is associated such that in the online phase that party broadcasts $d = x - r$ where x is its input to that wire; then, the parties perform a constant addition to obtain an authenticated share of x. See [12] for more details.

Denote by $btr(n), rnb(n), rnd(n), inp(n)$ the times to generate one beaver triple, one random bit, one random element, and entering one input element, respectively (which depend on the number of parties n). Let G_X and G_A be the number of XOR and AND gates in C_f, respectively. The preprocessing-I (SPDZ's offline phase) time is computed by

$$\begin{aligned} T_{\text{preprocessing-I}} &= \big(G_X(2n+3) + G_A(4n+5)\big) \cdot btr(n) \\ &\quad + B \cdot rnb(n) + R(n) \cdot rnd(n) + I(n) \cdot inp(n) \\ &= \big(G_X(2n+3) + G_A(4n+5)\big) \cdot btr(n) \\ &\quad + B \cdot rnb(n) + 2Wn \cdot rnd(n) + 8Gn \cdot inp(n). \end{aligned}$$

(Note that R and I also depend on n.)

The implementation of the SPDZ online phase, described in both [11,20], reports online throughputs of between 200,000 and 600,000 multiplications per second, depending on the system configuration. As remarked earlier, the online time of other operations

Table 2. Preprocessing times (in seconds) for the AES circuit.

No. parties	M	B	R	I	preprocessing-I	preprocessing-II
2	268,792	33,512	132,512	532,096	320	0.44–1.34
3	349,608	33,768	198,768	804,288	628	0.58–1.74
4	431,448	34,024	265,024	1,080,576	1640	0.71–2.15

is negligible and is therefore ignored. Thus, the preprocessing-II time is computed by

$$T_{\text{preprocessing-II}} = \frac{\big(G_X(2n+3) + G_A(4n+5)\big)}{mps},$$

where mps is the number of multiplications per second that the SPDZ system is able to perform.

To see what this would imply in practice, consider the AES circuit described in [29], which has become the standard benchmarking case for secure computation calculations. The basic AES circuit has $G \approx 33,000$ (with $G_A \approx 6000$ and $G_X \approx 27000$) gates and a similar number of wires W, including the key expansion within the circuit.[10] Assuming the parties share a XOR sharing of the AES key and data (which adds an additional $2 \cdot (n-1) \cdot 128$ gates and wires to the circuit), the parameters for the **Initialize** call to the $\mathcal{F}_{\text{SPDZ}}$ functionality in the preprocessing-I protocol will be

$$\begin{aligned} M &\approx (G_X + 256(n-1))(2n+3) + G_A(4n+5) \\ B &\approx G + 256n \\ R &\approx 2Wn + 256n \\ I &\approx 8 \cdot (G + 256(n-1)) \cdot n. \end{aligned}$$

Recall that M is the number of multiplications, B the number of random bits, R the number of random field elements, and I the number of input wires. Using the above execution times for the SPDZ protocol, we can then estimate the time needed for the two parts of our preprocessing step for the AES circuit. The expected execution times, in seconds, are given in Table 2.

These expected times, due to the methodology of our protocol, are likely to estimate both the latency and throughput amortized over many executions. (We only have these times for 2, 3, and 4 parties, since these are the times that have been published for SPDZ offline computations.)

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

[10] Note that unlike [29] and other Yao-based techniques we cannot process XOR gates for free. On the other hand, we are not restricted to only two parties.

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 runtimes for $n = 2, 3, 4$ parties were 6.35 ms, 9.88 ms, and 15 ms, respectively, for C code compiled using the gcc compiler on a 2.9 GHZ Xeon machine. The actual runtime, including all non-cryptographic operations, should be higher, but of the same order.

Our runtimes estimates compare favorably to several other results on implementing secure computation of AES in a multi-party setting:

- In [10], 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 [20], 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 [26], the authors report on a two-party MPC evaluation of the AES circuit using the TinyOT protocol; they obtain for 80 bits of security an amortized offline time of nearly 3 s per AES block, and an amortized online time of 30 ms; 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 connected by very fast connections, the effect of the number of rounds on the protocol will be extremely significant. For example, in [10], 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 ms (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 ms. We stress that even on a much faster network with a latency of just 10 ms, protocols with 50 rounds of communication will still be slow.

5. Security Proof

In this section, we prove the main theorem of this paper, i.e., see Theorem 1. The security proof contains two steps. In the first step, we reduce the security in the semi-honest case to the security of the original BMR protocol, that is, we only consider an adversary $\mathcal{A}$ that does not deviate from the prescribed protocol and only tries to learn information from the transcript. 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 showing a reduction from the malicious model to the semi-honest model. In both steps,

the adversary $\mathcal{A}$ is assumed to corrupt parties in the beginning of the execution of the protocol.

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 the bottom up until they reach the circuit-output wires. That is, the input wires are said to be at the "bottom" of the circuit, while the output wires are at the "top." In their evaluation, the players use the garbled 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 $\mathbf{k}_{c,\Lambda_c} = k^1_{c,\Lambda_c}, \ldots, k^n_{c,\Lambda_c}$. 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 entry $(2\Lambda_a + \Lambda_b)$ 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 denote the garbled entry for which the players evaluate that gate as the *active entry*. The other three entries are denoted as the *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 denotes their inputs. (Therefore, $|I \cup J| = n$ 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, respectively. We denote the set of gates in the circuit as $G = \{g_1, \ldots, g_{|G|}\}$. Recall that κ is the security parameter.

5.1. *Security in the Semi-honest Model*

The basic goal of the proof is to show that there exists 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,[11] namely $\text{REAL}^{\mathsf{BMR}}_{\mathcal{A}}$ 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 $\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})$ in View 2. Formally, the procedure is defined as

$$\mathcal{P} : \{\text{REAL}^{\mathsf{BMR}}_{\mathcal{A}}\}_{\bar{x}} \rightarrow \{\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})\}_{\bar{x}},$$

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

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

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 pseudorandom 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.

In this section, we present the procedure $\mathcal{P}$ and show that $\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{\bar{x}}$ and $\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})\}_{\bar{x}}$ 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 $\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})$ (View 2), then we describe the procedure $\mathcal{P}$ and prove the mentioned claims.

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

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 1), 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., $\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})$). We will then show that the resulting distribution of views is indistinguishable from $\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})$ 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 know $\bar{x}_J$ (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 input view.

Procedure 1. (The Procedure $\mathcal{P}$)

Input. A view v taken from distribution $\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}}$ under the input $\bar{x}$.

Output. A view v' conforming to the message flow in $\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})$. 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 garbled entries (except one as described immediately) to be random values from $(\mathbb{F}_p)^n$, while 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 constructs 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:

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

   for $j = 1, \ldots, n$ as described in Functionality 4. 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 1. *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

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

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

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

Then, if the following computational indistinguishability holds (proven in Claim 2)

$$\left\{\text{REAL}_{\mathcal{A}}^{\text{Our}}(\bar{x})\right\}_{\bar{x}} \stackrel{c}{\equiv} \left\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\text{BMR}})\right\}_{\bar{x}}, \tag{1}$$

then by transitivity of indistinguishability, it follows that

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

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 Eq. 1:

Claim 2. *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 procedure $\mathcal{P}$, both indexed by the players' inputs to the functionality $\bar{x}$, are computationally indistinguishable. That is:*

$$\left\{\text{REAL}_{\mathcal{A}}^{\text{Our}}(\bar{x})\right\}_{\bar{x}} \stackrel{c}{\equiv} \left\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\text{BMR}})\right\}_{\bar{x}}.$$

Proof. Remember that in procedure $\mathcal{P}$ we do not have any information about the masking values $\{\lambda_w \mid w \in W\}$ (except of those which are known to the adversary); therefore, we cannot compute the indicators x_A, x_B, x_C, x_D (as described in Sect. 4.2) and thus could not tell which key vector is encrypted in each entry. That is, we cannot fill out correctly the four garbled gate entries A, B, C, D. On the other hand, in procedure $\mathcal{P}$ we do know the set of external values $\{\Lambda_w \mid w \in W\}$; 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_cth key vector $\mathbf{k}_{c,\Lambda_c}$.

Let us denote by $\{\text{REAL}_{\mathcal{A}}^{\text{Our}}(\bar{x})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j}$ 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 $\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\text{BMR}})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j}$ the view of the adversary in the output of procedure $\mathcal{P}$.

Given that

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

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

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

since the functionality, keys, and masking values are taken from exactly the same distribution in both cases. □

In Claim 3, we prove that Eq. 2 holds.

Claim 3. *Fix a functionality f, the players' inputs $\bar{x}$, keys $\{k^i_{w,\beta} \mid 1 \leq i \leq n, w \in W, \beta \in \{0,1\}\}$ and masking values $\{\lambda_j \mid j \in W\}$ used in both the execution of our protocol and procedure $\mathcal{P}$, then Eq. (2) holds; that is,*

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

Proof. Remember that the difference between $\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j}$ and $\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j}$ is in the values of the entries of the garbled gates which are not in the active path, that is, in $\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j}$ these values are computed as described in Sect. 4.2 while in $\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j}$ they are just random values from $(\mathbb{F}_p)^n$.

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

$$\begin{aligned}|Pr[\mathcal{D}(\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j}) = 1] \\ - Pr[\mathcal{D}(\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j}) = 1]| = \varepsilon(\kappa)\end{aligned}$$

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

Let C be the Boolean circuit that computes the functionality f. For the purpose of the proof, we index the gates of C (the set of gates is denoted by G) in the following manner: C may be considered as a directed acyclic graph (DAG), where the gates are the nodes in the graph and a output wire of gate g_1 which enters as input wire to gate g_2 indicates the edge (g_1, g_2) in the graph. We compute a topological orderings of the graph, that is, if the output wire of gate g_1 enters to gate g_2, then the index that g_1 gets in the ordering is lower than the index of gate g_2. (Note that there might exist many valid topological ordering for the same graph.) For the sake of the proof, whenever we write g_i we refer to the i^{th} gate in the topological ordering.

We define the hybrid H^t as the view in which the gates $g_1, g_2, \ldots, g_t$ are computed as in procedure $\mathcal{P}$ (i.e., the inactive entries are just random elements from $(\mathbb{F}_p)^n$) and the gates $g_{t+1}, \ldots, g_{|G|}$ are computed as described in our protocol (Sect. 4.2). Observe that H^0 is distributed exactly as the view of the adversary in $\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j}$

and $H^{|G|}$ is distributed exactly as the view of the adversary in $\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\text{BMR}})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j}$. Thus, by a hybrid argument it follows that there exists an integer $0 \leq z < |G| - 1$ such that the distinguisher $\mathcal{D}$ 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) is computed as described in procedure $\mathcal{P}$. That is, the external values Λ_a, Λ_b, Λ_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, as written above, is an input wire to gate g); note that by the topological ordering of the gates the index of $\tilde{g}_a$ has a lower index than the index of g, and thus there is exactly one entry (the active entry) in the garbled table of $\tilde{g}_a$ which encrypts $\mathbf{k}_{a,\Lambda_a}$, while the other three (inactive) entries are random values from $(F_p)^n$ and therefore reveal no information about $\mathbf{k}_{a,\Lambda_a}$, and, more importantly, no information about $\mathbf{k}_{a,\bar{\Lambda_a}}$. The same observation holds for the gate $\tilde{g}_b$ whose output wire is b. Therefore, 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) involves 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 depend on at least one of $\mathbf{k}_{a,\bar{\Lambda_a}}$ and $\mathbf{k}_{b,\bar{\Lambda_b}}$, which the distinguisher $\mathcal{D}$ has no information about. Thus, 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 the view H^z), we could use some other key $\tilde{k}$ instead; in particular, we could use F without even knowing $\tilde{k}$ at all, e.g., when working with an oracle.

In the following analysis, we exploit this observation: Since the distinguisher $\mathcal{D}$ has no information about $\mathbf{k}_{a,\bar{\Lambda_a}}$ or $\mathbf{k}_{b,\bar{\Lambda_b}}$, we can construct the garbled table using some other keys, and because we are interested in the result of F under those keys (and not in the keys themselves) we can even use an oracle to replace a PRF. Thus, if $\mathcal{D}$ distinguishes between H^z and H^{z+1}, then we can use it to distinguish between an oracle to a pseudorandom function and an oracle to a truly random function (under multiple invocations of the oracle, because there are $2n$ keys in the two vectors $\mathbf{k}_{a,\bar{\Lambda_a}}$ and $\mathbf{k}_{b,\bar{\Lambda_b}}$).

Let us first define the notion of a pseudorandom 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 pseudorandom 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 pseudorandom 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 $m(n)$ ran-

dom functions from $\{0,1\}^n \to \{0,1\}^n$. 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 pseudorandom function, then it is a pseudorandom function under multiple keys. Thus, since the function F used in our protocol is a PRF, for every polynomial time distinguisher $\tilde{\mathcal{D}}$, every positive polynomial p and for all sufficiently large κ:

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

We 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 $\mathcal{D}$ which 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 $\overline{\mathcal{O}} = \mathcal{O}_1, \ldots, \mathcal{O}_{2n}$ (which is either a PRF under multiple keys or a set of truly random functions). $\mathcal{D}'$ acts 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 gates $g_1, \ldots, g_z$ as described in procedure $\mathcal{P}$, i.e., only the active entry is calculated correctly, while the other three entries are taken to be random from $(\mathbb{F}_p)^n$.
3. Constructs the garbled table of gate g_{z+1} in the following manner: Denote the input wires of the gate by a, b and the output wire by c; we want that the key vector $\mathbf{k}_{c,\Lambda_c}$ be encrypted using the key vectors $\mathbf{k}_{a,\Lambda_a}$ and $\mathbf{k}_{b,\Lambda_b}$ and held in the $2\Lambda_a + \Lambda_b$ entry. Thus:
   - Whenever a result of F applied to the key k^i_{a,Λ_a} is required, it is computed correctly as in the protocol. (The same holds for the key k^i_{b,Λ_b}.)
   - Whenever a result of F applied to the key $k^i_{a,\overline{\Lambda_a}}$ is required, the distinguisher $\mathcal{D}'$ queries the oracle $\mathcal{O}_i$ instead. (The same holds for the key $k^i_{b,\overline{\Lambda_b}}$; here, however, the distinguisher $\mathcal{D}'$ queries the oracle $\mathcal{O}_{n+i}$.)
4. Completes the computation of the garbled circuit, i.e., the garbled tables of gates $g_{z+2}, \ldots, g_{|G|}$, correctly, as in the 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:

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

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

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

□

5.2. *Security in the Malicious Model*

When our protocol relies on SPDZ as its underlying MPC, the keys that each party sees are guaranteed to be uniformly chosen from $\mathbb{F}_p$ and the masking values of all wires are guaranteed to be random values from $\{0, 1\}$. Thus, the garbled circuit is guaranteed to be built correctly and privately by the parties as a function of the original circuit C (which computes the functionality f), the set of keys of all parties, the set of masking values of all wires and by the PRF results that the parties apply to these keys. However, the elements of the last item (the PRF results) are not guaranteed to be computed correctly (moreover, below we show that it is wasteful to verify the correctness of their computation) and we must show that cheating in a PRF result(s) would cause the honest parties to abort.

Specifically, there are two locations in which a maliciously corrupted party might deviate from the protocol:

- A corrupted party might cheat in the offline phase by inputting a false value as one (or more) of the PRF results of its keys (i.e., PRF result that is not computed as described in 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 $\overline{\Lambda_w}$).

It is clear that the second 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 (i.e., a malicious adversary is able to change the input to the functionality without being considered as a cheat since this behavior is unavoidable even in the ideal model).

We break the proof of 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.

5.2.1. *Correctness*

Let us denote the event in which a corrupted party cheats by inputting a false PRF result in the offline phase as cheat (The event refers to *one* corrupted party and we show below that *even if only one party* cheats, then the honest parties abort). In the following, we prove the following claim:

Claim 4. *A malicious adversary cannot break the correctness property of our protocol except with negligible probability. Formally, denote the set of outputs of the honest parties in our protocol as* Π^J_{SFE} *and their outputs when computed by the functionality* f *as* y_J*,*

then for every positive polynomial p and for sufficiently large κ it holds that

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

Proof. To harm the correctness property of the protocol, the adversary has to provide to the offline phase incorrect results of F applied to its keys, such that the generated garbled circuit will cause the honest parties to output some set of values that is different from y_J.

Let GC_{SH} be the garbled circuit generated 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 random tape used by the underlying MPC, the adversary and the parties is the same, that is, same keys and masking values are being used).

Observe that if the adversary succeeds in breaking the correctness, then there must be at least one wire c and at least one honest party P_j where the gate g has input wires a, b and output wire c, such that in the evaluation of GC_{SH} (in the online phase) 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 $(\bar{v}, k^j_{c,\bar{v}})$ (that is, the adversary succeeded in flipping the signal that passes through wire c).

In the following analysis, we provide the adversary with 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.

We first analyze the success probability of the adversary in breaking 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 jth element of A_g is calculated (as described in Functionality 4) by

$$A^j_g = \left(\sum_{i=1}^{n} F_{k^i_{a,0}}(0 \,\|\, j \,\|\, g) + F_{k^i_{b,0}}(0 \,\|\, j \,\|\, g) \right) + k^j_{c,v}. \tag{4}$$

Recall that J is the set of corrupted parties and $J = [N] \setminus I$. For simplicity, define

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

$$= \sum_{i \in J} \left(F_{k^i_{a,0}}(0 \,\|\, j \,\|\, g) + F_{k^i_{b,0}}(0 \,\|\, j \,\|\, g) \right),$$

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 Eq. (4) we obtain

$$A^j_g = X^j + Y^j + k^j_{c,v}.$$

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

$$\tilde{A}^j_g = \tilde{X}^j + Y^j + k^j_{c,\bar{v}}.$$

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

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

as required and the jth 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 $k^j_{c,v}$ and $k^j_{c,\bar{v}}$ are random elements 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 of the garbled circuit. (In particular, they are independent of the keys $k^j_{c,v}$ and $k^j_{c,\bar{v}}$.) 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:

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

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 breaking 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:

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

for every positive polynomial q. □

5.2.2. *Emulation in the Ideal Model*

Next, we describe the ideal model in which the adversary's view will be emulated, and we show the existence of a simulator $\mathcal{S}'_{\text{OUR}}$ in the malicious model which uses 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 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 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 (as shown in [8]).

The ideal execution of f on inputs $\bar{x}$ and corrupted parties P_I (that are controlled by adversary $\mathcal{A}$) is denoted by $\text{IDEAL}^{f}_{\mathcal{A},I}(\bar{x})$ and the real execution is denoted by $\text{REAL-MAL}^{\mathsf{Our}}_{\mathcal{A},I}(\bar{x})$; in both cases, they refer to the joint distribution of the outputs of *all* parties. (In the following proof, we use $\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})$ to refer to the real execution in the semi-honest model.)

Proof outline. In the following proof, we make use of two procedures, $\mathcal{P}'$ (which is close to procedure $\mathcal{P}$) and $\mathcal{H}$. The procedure $\mathcal{P}'$ is given a view of the adversary in the semi-honest model (or a view that is indistinguishable from it, e.g., a simulated view) and a set of keys $\mathbf{K_I}$, and outputs the exact same view, with the exception that the keys that are opened to the adversary are now $\mathbf{K_I}$. The procedure $\mathcal{H}$ is given a view of the adversary in the semi-honest model (or a view that is indistinguishable from it) and a set of PRF results $\mathbf{F_I}$, and outputs the exact same view, with the exception that it applies the set of PRF results $\mathbf{F_I}$ to the view as if the adversary has provided them in the real execution of the protocol (that is, the set $\mathbf{F_I}$ affects the exact same locations in the input view that it would have affect it in a real execution of the protocol in the malicious model).

The simulator $\mathcal{S}'_{\text{OUR}}$ 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 $\mathcal{S}'_{\text{OUR}}$ also instructs 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 from their output in the real execution of our protocol.

The idea of the simulation method is that we can use the fact that there exists a simulator $\mathcal{S}_{\text{OUR}}$ in the semi-honest model which can construct a garbled circuit that is indistinguishable from the one constructed by our protocol in the semi-honest model. By internally running $\mathcal{A}$, the simulator $\mathcal{S}'_{\text{OUR}}$ can extract the inputs of the adversary $\bar{x}$, the keys $\mathbf{K_I}$ that were opened to it and the exact locations in which $\mathcal{A}$ has cheated (that is, the set $\mathbf{F_I}$ of PRF results that it provides given that the set of keys that it sees are

$\mathbf{K_I}$). Hence, using the procedures $\mathcal{P}'$ and $\mathcal{H}$, the simulator $\mathcal{S}'_{\text{OUR}}$ can tweak the garbled circuit which resulted by $\mathcal{S}_{\text{OUR}}$ in the specific locations to match the garbled circuit.

The procedure $\mathcal{P}'$. Let us define the procedure $\mathcal{P}'$ (Procedure 2) which receives as input a view simulated by $\mathcal{S}_{\text{OUR}}$ or a real view of the adversary in the semi-honest model ($\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})$), along with a set of keys $\mathbf{K_I} = \{k^i_{w,j} \mid i \in I, w \in W, j \in \{0,1\}\}$ (i.e., two keys per wire per corrupted party) and rebuilds the garbled circuit just as $\mathcal{P}$ did (in the semi-honest case), but instead of using random keys of its choice it uses the keys received as input for the corrupted parties I. Even though Procedure $\mathcal{P}$ was originally used to transform a view of the BMR execution into a view of the execution of our protocol, we can use it to transform a view of our protocol into another view of our protocol (e.g., by only changing the keys); this is exactly what we do in the simulation.

Procedure 2. (The Procedure $\mathcal{P}'$)

Input.

- A view v taken from distribution $\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})$ or the output of $\mathcal{S}_{\text{OUR}}$.
- 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' which is the same as v, but in v the garbled circuit is built using the set of keys $\mathbf{K_I}$ from the input.
Execute procedure $\mathcal{P}$ on v with the exception that in step 3a use the keys $\mathbf{K_I}$ given as input rather than choosing new ones for every key of parties I.

Claim 5. *Denote by* $\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})(\bar{x})$ *the view of the semi-honest adversary in our protocol when the inputs of the parties are* $\bar{x}$*, and denote by* $\mathcal{P}'(\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})(\bar{x}), \mathbf{K_I})$ *the result of procedure* $\mathcal{P}'$ *applied on* $\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})(\bar{x})$ *using the keys* $\mathbf{K_I}$*; then, given that the keys in* $\mathbf{K_I}$ *are chosen uniformly from* $\mathbb{F}_p$ *it follows that for every* $\bar{x}$

$$\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x}) \stackrel{c}{\equiv} \mathcal{P}'(\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x}), \mathbf{K_I}) \tag{5}$$

Proof. The proof follows identically the proof of Claim 2. □

Corollary 5.1. *Given that the keys in* $\mathbf{K_I}$ *are chosen uniformly from* $\mathbb{F}_p$*, the probability ensemble of the view in the semi-honest model* $\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})$ *and the view when procedure* $\mathcal{P}'$ *applied on it (using* $\mathbf{K_I}$*), such that the ensembles are indexed by the inputs of the parties* $\bar{x}$*, are indistinguishable. That is*

$$\left\{\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})\right\}_{\bar{x}} \stackrel{c}{\equiv} \left\{\mathcal{P}'(\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x}), \mathbf{K_I})\right\}_{\bar{x}}. \tag{6}$$

Procedure 3. (The Procedure $\mathcal{H}$)

Input. A view v taken from distribution $\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})$ under the input $\bar{x}$; and a set of PRF results $\mathbf{F_I}$ of F applied to the set of keys of parties $\{P_i\}_{i\in I}$ (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 $\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})$ 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 $\mathbf{F_I}$ are computed correctly and which are 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 follows (where g is the gate to which wire w enters):

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

while the correct PRF values as:

$$\{F_{k^i_{w,b}}(0\|1\|g), \ldots, F_{k^i_{w,b}}(0\|n\|g)\}_{i\in I, w\in W, b\in\{0,1\}}$$
$$\{F_{k^i_{w,b}}(1\|1\|g), \ldots, F_{k^i_{w,b}}(1\|n\|g)\}_{i\in I, w\in W, b\in\{0,1\}}$$

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 4, 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 $F_{k^i_{w,b}}(\alpha \parallel \beta \parallel \gamma)$. Then, it computes the difference

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

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 $F^{\Delta}_{k^i_{a,0}}(0 \parallel j \parallel g) = \tilde{F}_{k^i_{a,0}}(0 \parallel j \parallel g) - F_{k^i_{a,0}}(0 \parallel j \parallel g)$, then the procedure adds $F^{\Delta}_{k^i_{a,0}}(0 \parallel j \parallel g)$ to the value A_g given in v.

When done with those changes, the procedure outputs the resulted view v'.

The procedure $\mathcal{H}$ We now define the procedure $\mathcal{H}$ (Procedure 3) which is given a view from the distribution $\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})$ and a set of PRF results $\mathbf{F_I}$ (computed correctly or not) for every key of parties $\{P_i\}_{i\in I}$. 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 inputs 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) depend only on the adversary's random tape r, and on the keys and masking values outputted to them from the underlying MPC. 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 Procedure $\mathcal{H}$; we get the following:

Claim 6. *Let* $\text{REAL-MAL}^{\mathsf{Our}}_{\mathcal{A},I}(\bar{x})_{\mathbf{K_I},\mathbf{F_I}}$ *be the view of the adversary (not the joint view of all parties) in the execution of our protocol in the malicious model where the keys that the adversary sees are* $\mathbf{K_I}$*, and the PRF results that it provides are* $\mathbf{F_I}$*. Similarly, let* $\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})_{\mathbf{K_I}}$ *be the view of the adversary in the execution of our protocol in the semi-honest model where the keys that it sees are* $\mathbf{K_I}$*. For every* $\{\mathbf{K_I}, \mathbf{F_I}\}$*, it follows that*

$$\begin{aligned}\text{REAL-MAL}^{\mathsf{Our}}_{\mathcal{A},I}(\bar{x})_{\mathbf{K_I},\mathbf{F_I}} &\equiv \mathcal{H}(\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})_{\mathbf{K_I}}, \mathbf{F_I}) \\ &= \mathcal{H}(\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})_{\mathbf{K_I}}, \mathcal{A}(r, \mathbf{K_I}, \lambda_\mathbf{I})). \end{aligned} \tag{7}$$

Proof. The proof follows immediately from the definition of the procedure $\mathcal{H}$. □

The simulator $\mathcal{S}'_{\text{OUR}}$ As mentioned earlier, the simulator $\mathcal{S}'_{\text{OUR}}$ uses the procedures $\mathcal{H}$ and $\mathcal{P}'$ described above:

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 halts the internal execution right after it receives the external values $\Lambda_\mathbf{I}$ to all the corrupted parties in the online phase (that is, it halts after Step 4
   of the online phase of Protocol 2). From the internal execution, the simulator $\mathcal{S}'_{\text{OUR}}$ can extract (learn) the following values:
   (a) The keys $k^I_{w,0}, k^I_{w,1}$ (of the adversary, 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.
   (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 results (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 $\Lambda_\mathbf{I}$, 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 was 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. The simulator $\mathcal{S}'_{\text{OUR}}$ computes $v' = \mathcal{P}'(v, \mathbf{K_I})$.

6. The simulator $\mathcal{S}'_{\text{OUR}}$ 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 with the inputs x_I and $x_J = 0^{|J|}$.[12] If these parties 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.[13]
8. The simulator $\mathcal{S}'_{\text{OUR}}$ outputs the view v'' as the adversary's simulated output.

Indistinguishability: Real Versus Ideal. To complete the proof of security in the malicious model, we have to prove the following:

Claim 7. *The distribution ensemble of the output of the parties under the simulation of $\mathcal{S}'_{OUR}$ and under the real execution of our protocol are indistinguishable.*

Formally, let $\{\text{REAL-MAL}^{\mathsf{Our}}_{\mathcal{A},I}(\bar{x})\}_{\bar{x}}$ be the probability ensemble (indexed by the inputs of the parties) of the view of the parties that are under the control of the adversary $\mathcal{A}$ in the real execution of our protocol and $\{\text{IDEAL}^{\mathcal{S}'_{OUR}}_{\mathcal{A}}(\bar{x})\}_{\bar{x}}$ be the probability ensemble of their view in the execution aided by a trusted party (i.e., in the ideal model with the simulator $\mathcal{S}'_{OUR}$), then:

$$\left\{\text{REAL-MAL}^{\mathsf{Our}}_{\mathcal{A},I}(\bar{x})\right\}_{\bar{x}} \stackrel{c}{\equiv} \left\{\text{IDEAL}^{\mathcal{S}'_{OUR}}_{\mathcal{A}}(\bar{x})\right\}_{\bar{x}}.$$

Proof. Immediate from the proof of Claim 8. That is, in Claim 8 we state the same thing and prove it for every possible set of inputs of the players $\bar{x}$. □

Claim 8. *For every $\bar{x}$, it holds that*

$$\text{REAL-MAL}^{\mathsf{Our}}_{\mathcal{A},I}(\bar{x}) \stackrel{c}{\equiv} \text{IDEAL}^{\mathcal{S}'_{OUR}}_{\mathcal{A}}(\bar{x}).$$

Proof. Let $V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x})$ be the view of the adversary in the real execution of our protocol (i.e., the view of the adversary that is taken from $\text{REAL-MAL}^{\mathsf{Our}}_{\mathcal{A},I}(\bar{x})$) and $V^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x})$ be the view of the adversary that the simulator $\mathcal{S}_{\text{OUR}}{}'$ outputs; also, let $O^{J}_{\text{REAL-MAL}}(\bar{x})$ be the output of the honest parties in the real execution of the protocol and $O^{J,\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x})$ be their output in the ideal model.

We can obviously restate our claim as:

$$\left\{V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x}),\, O^{J}_{\text{REAL-MAL}}(\bar{x})\right\} \stackrel{c}{\equiv} \left\{V^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x}),\, O^{J,\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x})\right\}.$$

[12] Note that the correctness property has shown earlier holds for every input of the honest parties x_J; thus, in order to decide whether to instruct the trusted party to "halt" or "continue" $\mathcal{S}'_{\text{OUR}}$ can just use some fake input $x_J = 0^{|J|}$.

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

Given that $V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x}) \stackrel{c}{\equiv} V^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x})$ (which is proven in Claim 9) we now prove the above by a reduction. Assume by contradiction that there exist a PPT distinguisher $\mathcal{D}$ and a non-negligible function ε in κ such that

$$\begin{aligned}|Pr[\mathcal{D}(\{V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x}), O^{J}_{\text{REAL-MAL}}(\bar{x})\}) = 1] \\ - Pr[\mathcal{D}(\{V^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x}), O^{J,\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x})\}) = 1]| = \varepsilon(\kappa).\end{aligned}$$

We describe a distinguisher $\mathcal{D}'$ that is able to distinguish between $V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x})$ and $V^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x})$ with non-negligible probability; note that since we prove the above for every choice of $\bar{x}$ the distinguisher may use $\bar{x}$ in its algorithm. The distinguisher $\mathcal{D}'$ acts as follows:

1. The distinguisher $\mathcal{D}'$ is given a view v of the adversary which is either from a real execution of the protocol or a simulated view, i.e., either $V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x})$ or $V^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x})$.
2. The view v contains the garbled circuit constructed either by the players or by the simulator; moreover, as mentioned above, $\mathcal{D}'$ knows the inputs of all parties (because we prove the claim for specific choice of $\bar{x}$); thus, $\mathcal{D}'$ evaluates the circuit using $\bar{x}$ and assigns the output of the honest parties into y_J.
3. The distinguisher $\mathcal{D}'$ hands $\{v, y_J\}$ to $\mathcal{D}$ and outputs whatever it outputs.

From the correctness property shown in the proof of Claim 4, it follows that if v has been taken from $V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x})$, then $\{v, y_J\}$ and $\{V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x}), O^{J}_{\text{REAL-MAL}}(\bar{x})\}$ are indistinguishable; otherwise, if v has been taken from $V^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x})$, then $\{v, y_J\}$ and $\{V^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x}), O^{J,\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x})\}$ are indistinguishable due to the simple fact that the distinguisher $\mathcal{D}'$ does exactly what the honest parties do in the real execution. Formally:

$$\begin{aligned}|Pr[\mathcal{D}(\{V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x}), O^{J}_{\text{REAL-MAL}}(\bar{x})\}) = 1] \\ - Pr[\mathcal{D}'(V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x})) = 1]| = \varepsilon_2(\kappa) \\ |Pr[\mathcal{D}(\{V^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x}), O^{J,\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x})\}) = 1] \\ - Pr[\mathcal{D}'(V^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x})) = 1]| = \varepsilon_3(\kappa),\end{aligned}$$

where $\varepsilon_2(\kappa)$ and $\varepsilon_3(\kappa)$ are negligible. It follows that

$$\begin{aligned}&Pr[\mathcal{D}'(V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x})) = 1] \\ &\quad = Pr[\mathcal{D}(\{V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x}), O^{J}_{\text{REAL-MAL}}(\bar{x})\}) = 1] - \varepsilon_2(\kappa) \quad \text{and} \\ &Pr[\mathcal{D}'(V^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x})) = 1] \\ &\quad = Pr[\mathcal{D}(\{V^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x}), O^{J,\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x})\}) = 1]| - \varepsilon_3(\kappa)\end{aligned}$$

and thus

$$Pr[\mathcal{D}'(V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x})) = 1] - Pr[\mathcal{D}'(V^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x})) = 1]$$

$$= \varepsilon(\kappa) - \varepsilon_2(\kappa) + \varepsilon_3(\kappa),$$

which is non-negligible, in contradiction to the result in Claim 9. □

Claim 9. *Let $V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x})$ be the view of the adversary in the real execution of our protocol and $V^{\mathcal{A},\mathcal{S}'_{OUR}}_{\text{IDEAL}}(\bar{x})$ be the view of the adversary outputted by the simulator $\mathcal{S}_{OUR}{}'$ such that in both cases the inputs to the protocol are $\bar{x}$.*

For every $\bar{x}$, it holds that

$$V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x}) \stackrel{c}{\equiv} V^{\mathcal{A},\mathcal{S}'_{OUR}}_{\text{IDEAL}}(\bar{x}).$$

Proof. From the above definitions of Procedure $\mathcal{P}'$ and $\mathcal{H}$, we get:

$$\begin{aligned}
\text{REAL-MAL}^{\mathsf{Our}}_{\mathcal{A},I}(\bar{x})_{\mathbf{K_I},\mathbf{F_I}} &\equiv \mathcal{H}(\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})_{\mathbf{K_I}}, \mathbf{F_I}) \\
&\stackrel{c}{\equiv} \mathcal{H}(\mathcal{P}'(\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})_{\mathbf{K_I}}), \mathbf{F_I}) \\
&\stackrel{c}{\equiv} \mathcal{H}(\mathcal{P}'(\mathcal{S}_{\text{OUR}}(1^\kappa, I, x_I, y_I)_{\mathbf{K_I}}), \mathbf{F_I}).
\end{aligned}$$

Where the first equality is given from Eq. 7, the second follows from 5 and the third follows from the operation of the simulator of the semi-honest model. That is, if there exist a distinguisher who succeeds to distinguish between $V^{\mathcal{A}}_{\text{REAL-MAL}}(\bar{x})$ and $V^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}_{\text{IDEAL}}(\bar{x})$ with non-negligible probability, then we can easily construct a distinguisher who is able to distinguish between $\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})$ and $\mathcal{S}_{\text{OUR}}(1^\kappa, I, x_I, y_I)$ in contradiction to the security in the semi-honest model. □

Acknowledgements

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 no. 615172 (HIPS). The second author was supported under the European Union's Seventh Framework Program (FP7/2007-2013) grant agreement no. 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, by EPSRC via grant EP/I03126X and by ERC Advanced Grant ERC-2015-AdGIMPaCT. 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. The first, second and fourth authors were supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Directorate in the Prime Minister's Office.

Appendix: 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 Sect. 4, 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 $[\mathbf{k}_{w,j}]$.

References

[1] D. Beaver, S. Micali, P. Rogaway, The round complexity of secure protocols, in *22nd STOC*, pp. 503–513, 1990

[2] A. Ben-David, N. Nisan, B. Pinkas, M. P. Fairplay, A system for secure multi-party computation, in *ACM CCS*, pp. 257–266, 2008

[3] A. Ben-Efraim, Y. Lindell, E. Omri, Optimizing semi-honest secure multiparty computation for the internet, in *ACM CCS*, pp. 578–590, 2016

[4] A. Ben-Efraim, Y. Lindell, E. Omri, Efficient scalable constant-round MPC via garbled circuits, in *ASIACRYPT*, pp. 471–498, 2017

[5] M. Ben-Or, S. Goldwasser, A. Wigderson, Completeness theorems for non-cryptographic fault-tolerant distributed computation, in *20th STOC*, pp. 1–10, 1988

[6] D. Chaum, C. Crépeau, I. Damgård, Multiparty unconditionally secure protocols, in *20th STOC*, pp. 11–19, 1988

[7] S. G. Choi, J. Katz, A. J. Malozemoff, V. Zikas, Efficient three-party computation from cut-and-choose, in *CRYPTO*, pp. 513–530, 2014

[8] R. Cleve, Limits on the security of coin flips when half the processors are faulty (extended abstract), in *18th STOC*, pp. 364–369, 1986
[9] I. Damgård, Y. Ishai, Constant-round multiparty computation using a black-box pseudorandom generator, in *CRYPTO*, pp. 378–394, 2005
[10] I. Damgård, M. Keller, E. Larraia, C. Miles, N. P. Smart, Implementing AES via an actively/covertly secure dishonest-majority MPC protocol, in *SCN*, pp. 241–263, 2012
[11] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, N. P. Smart, Practical covertly secure MPC for dishonest majority—or: Breaking the SPDZ limits, in *ESORICS*, pp. 1–18, 2013
[12] I. Damgård, V. Pastro, N. P. Smart, S. Zakarias, Multiparty computation from somewhat homomorphic encryption, in *CRYPTO*, pp. 643–662, 2012
[13] P. Feldman, S. Micali, An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM Journal on Computing* **26**(4), 873–933 (1997)
[14] O. Goldreich, S. Micali, A. Wigderson, How to play any mental game or A completeness theorem for protocols with honest majority, in *19th STOC*, pp. 218–229, 1987
[15] S. Goldwasser, Y. Lindell, Secure computation without agreement, in *DISC*, pp. 17–32, 2002
[16] C. Hazay, P. Scholl, E. Soria-Vazquez, Low cost constant round MPC combining BMR and oblivious transfer, in *ASIACRYPT*, pp. 598–628, 2017
[17] Y. Ishai, M. Prabhakaran, A. Sahai, Founding cryptography on oblivious transfer—efficiently, in *CRYPTO*, pp. 572–591, 2008
[18] J. Katz, R. Ostrovsky, A. D. Smith, Round efficiency of multi-party computation with a dishonest majority, in *EUROCRYPT*, pp. 578–595, 2003
[19] M. Keller, E. Orsini, P. Scholl, MASCOT: faster malicious arithmetic secure computation with oblivious transfer, in *ACM CCS, 2016*, pp. 830–842, 2016
[20] M. Keller, P. Scholl, N. P. Smart, An architecture for practical actively secure MPC with dishonest majority, in *ACM CCS*, pp. 549–560, 2013
[21] M. Keller, A. Yanai, Efficient maliciously secure multiparty computation for RAM, in *EUROCRYPT, 2018*, pp. 91–124, 2018
[22] E. Larraia, E. Orsini, N. P. Smart, Dishonest majority multi-party computation for binary circuits, in *CRYPTO, 2014*, pp. 495–512, 2014
[23] Y. Lindell, Fast cut-and-choose based protocols for malicious and covert adversaries, in *CRYPTO*, pp. 1–17, 2013
[24] Y. Lindell, B. Riva, Cut-and-choose yao-based secure computation in the online/offline and batch settings, in *CRYPTO*, pp. 476–494, 2014
[25] Y. Lindell, N. P. Smart, E. Soria-Vazquez, More efficient constant-round multi-party computation from BMR and SHE, in *14th TCC 2016-B*, pp. 554–581, 2016
[26] J. B. Nielsen, P. S. Nordholt, C. Orlandi, S. S. Burra, A new approach to practical active-secure two-party computation, in *CRYPTO*, pp. 681–700, 2012
[27] R. Pass, Bounded-concurrent secure multi-party computation with a dishonest majority, in *36th STOC*, pp. 232–241, 2004
[28] M. C. Pease, R. E. Shostak, L. Lamport, Reaching agreement in the presence of faults. *Journal of ACM* **27**(2), 228–234 (1980)
[29] B. Pinkas, T. Schneider, N. P. Smart, S. C. Williams, Secure two-party computation is practical, in *ASIACRYPT*, pp. 250–267, 2009
[30] T. Rabin, M. Ben-Or, Verifiable secret sharing and multiparty protocols with honest majority, in *21st STOC*, pp. 73–85, 1989
[31] P. Rogaway, *The round complexity of secure protocols*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1991
[32] A. C. Yao, Protocols for secure computations, in *23rd FOCS*, pp. 160–164, 1982