

# Efficient Constant Round Multi-Party Computation Combining BMR and SPDZ

Yehuda Lindell<sup>1</sup>, Benny Pinkas<sup>1</sup>, Nigel P. Smart<sup>2</sup>, and Avishay Yanai<sup>1</sup>

<sup>1</sup> Dept. Computer Science, Bar-Ilan University, Israel,

<sup>2</sup> Dept. Computer Science, University of Bristol, UK

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

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

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

## 1 Introduction

**Background:** Protocols for secure multi-party computation (MPC) enable a set of mutually distrustful parties to securely compute a joint functionality of their inputs. Such a protocol must guarantee *privacy* (meaning that only the output is learned), *correctness* (meaning that the output is correctly computed from the inputs), and *independence of inputs* (meaning that each party must choose its input independently of the others). Formally, security is defined by comparing the distribution of the outputs of all parties in a real protocol to an ideal model where an incorruptible trusted party computes the functionality for the parties. The two main types of adversaries that have been considered are *semi-honest adversaries* who follow the protocol specification but try to learn more than allowed by inspecting the transcript, and *malicious adversaries* who can run any arbitrary strategy in an attempt to break the protocol. Secure MPC has been studied since the late 1980s, and powerful feasibility results were proven showing that *any* two-party or multi-party functionality can be securely computed [22, 10], even in the presence of malicious adversaries. When an honest majority

(or 2/3 majority) is assumed, then security can even be obtained information theoretically [3, 4, 19]. In this paper, we focus on the problem of security in the presence of malicious adversaries, and a dishonest majority.

Recently, there has been much interest in the problem of concretely efficient secure MPC, where “concretely efficient” refers to protocols that are sufficiently efficient to be implemented in practice (in particular, these protocols should only make black-box usage of cryptographic primitives; they must not, say, use generic ZK proofs that operate on the circuit representation of these primitives). In the last few years there has been tremendous progress on this problem, and there now exist extremely fast protocols that can be used in practice; see [14, 16, 17, 13, 8] for just a few examples. In general, there are two approaches that have been followed; the first uses Yao’s garbled circuits [22] and the second utilizes interaction for every gate like the GMW protocol [10].

There are extremely efficient variants of Yao’s protocol for the two party case that are secure against malicious adversaries (e.g., [14, 16]). These protocols run in a constant number of rounds and therefore remain fast over slow networks. The BMR protocol [1] is a variant of Yao’s protocol that runs in a multi-party setting with more than two parties. This protocol works by the parties jointly constructing a garbled circuit (possibly in an offline phase), and then later computing it (possibly in an online phase). However, in the case of malicious adversaries this protocol suffers from two main drawbacks: (1) Security is only guaranteed if at most a *minority* of the parties are corrupt; (2) The protocol uses generic protocols secure against malicious adversaries (say, the GMW protocol) that evaluate the pseudorandom generator used in the BMR protocol. This non black-box construction results in an extremely high overhead.

The TinyOT and SPDZ protocols [17, 8] follow the GMW paradigm, and have offline and online phases. Both of these protocols overcome the issues of the BMR protocol in that they are secure against any number of corrupt parties, make only black-box usage of cryptographic primitives, and have very fast online phases that require only very simple (information theoretic) operations. (A black-box constant-round MPC construction appears in [11]; however, it is not “concretely efficient”.) In the case of multi-party computation with more than two parties, these protocols are currently the *only practical* approach known. However, since they follow the GMW paradigm, their online phase requires a communication round for every multiplication gate. This results in a large amount of interaction and high latency, especially over slow networks. To sum up, there is no known concretely efficient constant-round protocol for the multi-party case (with the exception of [5] that considers the specific three-party case only). Our work introduces the first protocol with these properties.

**Our contribution:** In this paper, we provide the first *concretely efficient constant-round* protocol for the general *multi-party* case, with security in the presence of malicious adversaries. The basic idea behind the construction is to use an efficient non-constant round protocol – with security for malicious adversaries – to compute the gate tables of the BMR garbled circuit (and since the computation of these tables is of constant depth, this step is constant round). A crucial

observation, resulting in a great performance improvement, shows that in the offline stage it is *not required to verify the correctness* of the computations of the different tables. Rather, validation of the correctness is an immediate by product of the online computation phase, and therefore does not add any overhead to the computation. Although our basic generic protocol can be instantiated with any non-constant round MPC protocol, we provide an optimized version that utilizes specific features of the SPDZ protocol [8].

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

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

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

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

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

## 2 The General Protocol

### 2.1 The BMR Protocol

To aid the reader we provide here a high-level description of the BMR protocol of [1]. A detailed description of the protocol can be found in [1, 2] or in the full version of our paper [15]. We describe here the version of the protocol that is secure against semi-honest adversaries. The protocol is comprised of an **offline-phase**, where the garbled circuit is created by the players, and an **online-phase**, where garbled inputs are exchanged between the players and the circuit is evaluated.

**Seeds and superseeds:** Each player associates random 0-seed and 1-seed with each wire. Input wires of the circuit are treated differently, and there only the player which provides the corresponding input bit knows the seeds of the wire. The 0-superseed (resp. 1-superseed) of a wire is the concatenation of all 0-seeds (1-seeds) of this wire, and its *components* are the seeds.

**Garbling:** For each of the four combinations of input values to a gate, the garbling produces an encryption of the corresponding superseed of the output wire, with the keys being each of the component seeds of the corresponding superseeds of the input wires.

**The offline phase:** In the **offline-phase**, the players run (in parallel) a secure computation for each gate, which computes the garbled table of the gate as a function of the 0/1-seeds of each of the players for the input/output wires of the gate, and of the truth table of the gate. This computation runs in a constant number of rounds. The resulting garbled table enables to compute the superseed of the output wire of the gate, given the superseeds of its input wires.

**The online phase:** In the **online-phase** each player which is assigned an input wire, and which has an input value  $b$  on that wire, sends the  $b$ -superseed of the wire to all other players. Then, every player is able evaluate the circuit on its own, without any further interaction with the other players.

### 2.2 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 above. First,

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

Second, we observe that the external values<sup>3</sup> 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  $\Lambda$ ) 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 means that the pseudorandom function is not treated as a black box, and the circuit for the offline phase would be huge (as it would include multiple copies of a subcircuit for computing pseudorandom function computations for every wire). Instead, we prove that this type of corrupt behavior can only result in an abort in the online phase, which would not affect the security of the protocol. This observation enables us to compute each gate independently and model the pseudorandom function used in the computation as a black box, thus simplifying the protocol and optimizing its performance.

We also encrypt garbled values as *vectors*; this enables us to use a finite field that can encode  $\{0, 1\}^\kappa$  (for each vector coordinate), rather than a much larger finite field that can encode all of  $\{0, 1\}^{n \cdot \kappa}$ . Due to this, the parties choose *keys* (for a pseudorandom function) rather than *seeds* for a pseudorandom generator. The keys that  $P_i$  chooses for wire  $w$  are denoted  $k_{w,0}^i$  and  $k_{w,1}^i$ , 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  $2^\kappa$ , 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  $\kappa$  bits, as  $F_k(x) = \text{CBC-MAC}_{\text{enc}}(k \pmod{2^\kappa}, x)$ . Note that the inputs  $x$  to

---

<sup>3</sup> 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.

our pseudorandom function will all be of the same length and so using naive CBC-MAC will be secure.

We interpret the  $\kappa$ -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  $\kappa$ -bit block cipher key by computing  $k \pmod{2^\kappa}$  induces a distribution on the key space of the block cipher which has statistical distance from uniform of

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

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

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

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

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

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

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

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

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

**Output:** On input  $(output)$  from all honest parties the functionality computes  $y = f(x_1, \dots, x_n)$  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  $\Pi_{\text{SFE}}$  which implements the Secure Function Evaluation (SFE) functionality of Functionality 1 in a constant number of rounds in the case of a malicious dishonest majority. Our constant round protocol  $\Pi_{\text{SFE}}$  implementing  $\mathcal{F}_{\text{SFE}}$  is built in the  $\mathcal{F}_{\text{MPC}}$ -hybrid model, i.e. utilizing a sub-protocol  $\Pi_{\text{MPC}}$  which implements the functionality  $\mathcal{F}_{\text{MPC}}$  given in Functionality 2. The generic MPC functionality  $\mathcal{F}_{\text{MPC}}$  is *reactive*. We require a *reactive* MPC functionality because our protocol  $\Pi_{\text{SFE}}$  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  $\Pi_{\text{MPC}}$  is

that each of the commands which it implements can be implemented in constant rounds. Given this requirement our larger protocol  $\Pi_{\text{SFE}}$  will be constant round.

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

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

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

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

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

**Add:** On command  $(add, varid_1, varid_2, varid_3)$  from all parties (if  $varid_1, varid_2$  are present in memory and  $varid_3$  is not), the functionality retrieves  $(varid_1, x)$ ,  $(varid_2, y)$  and stores  $(varid_3, x + y \bmod p)$ . The functionality then calls **Wait**.

**Multiply:** On input  $(multiply, varid_1, varid_2, varid_3)$  from all parties (if  $varid_1, varid_2$  are present in memory and  $varid_3$  is not), the functionality retrieves  $(varid_1, x)$ ,  $(varid_2, y)$  and stores  $(varid_3, x \cdot y \bmod p)$ . The functionality then calls **Wait**.

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

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

### 2.3 The Offline Functionality: preprocessing-I and preprocessing-II

Our protocol,  $\Pi_{\text{SFE}}$ , is comprised of an **offline-phase** and an **online-phase**, where the **offline-phase**, which implements the functionality  $\mathcal{F}_{\text{offline}}$ , is divided into two subphases: **preprocessing-I** and **preprocessing-II**. To aid exposition we first present the functionality  $\mathcal{F}_{\text{offline}}$  in Functionality 3. In the next section, we present an efficient methodology to implement  $\mathcal{F}_{\text{offline}}$  which uses the SPDZ protocol as the underlying MPC protocol for securely computing functionality  $\mathcal{F}_{\text{MPC}}$ ; while in the full version of the paper [15] we present a generic implementation of  $\mathcal{F}_{\text{offline}}$  based on any underlying protocol  $\Pi_{\text{MPC}}$  implementing  $\mathcal{F}_{\text{MPC}}$ .

In describing functionality  $\mathcal{F}_{\text{offline}}$  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  $\ell$  inputs to the functionality  $f$  then there are  $n \cdot \ell$  attached wires. The rest of the wires are considered as *common* wires, i.e. they are directly connected to *none* of the parties.

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

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

## 2.4 Securely Computing $\mathcal{F}_{\text{SFE}}$ in the $\mathcal{F}_{\text{offline}}$ -Hybrid Model

We now define our protocol  $\Pi_{\text{SFE}}$  for securely computing  $\mathcal{F}_{\text{SFE}}$  (using the BMR garbled circuit) in the  $\mathcal{F}_{\text{offline}}$ -hybrid model, see Protocol 1. In the full version of this paper [15], we prove the following theorem:

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

## 2.5 Implementing $\mathcal{F}_{\text{offline}}$ in the $\mathcal{F}_{\text{MPC}}$ -Hybrid Model

At first sight, it may seem that in order to construct an entire garbled circuit (i.e. the output of  $\mathcal{F}_{\text{offline}}$ ), 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 may input inconsistent values. For example, the masking values  $\lambda_w$  that are common to a number of gates (which happens when any wire

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

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

**preprocessing-I:** On input (preprocessing-I,  $W, G$ ), for all wires  $w \in [1, \dots, W]$ :

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

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

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

$$\begin{array}{ll} F_{k_{x,0}^i}(0\|1\|g), \dots, F_{k_{x,0}^i}(0\|n\|g) & F_{k_{x,0}^i}(1\|1\|g), \dots, F_{k_{x,0}^i}(1\|n\|g) \\ F_{k_{x,1}^i}(0\|1\|g), \dots, F_{k_{x,1}^i}(0\|n\|g) & F_{k_{x,1}^i}(1\|1\|g), \dots, F_{k_{x,1}^i}(1\|n\|g) \end{array}$$

- Define the selector variables

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

- Set  $\mathbf{A}_g = (A_g^1, \dots, A_g^n)$ ,  $\mathbf{B}_g = (B_g^1, \dots, B_g^n)$ ,  $\mathbf{C}_g = (C_g^1, \dots, C_g^n)$ , and  $\mathbf{D}_g = (D_g^1, \dots, D_g^n)$  where for  $1 \leq j \leq n$ :

$$\begin{aligned} A_g^j &= \left( \sum_{i=1}^n F_{k_{a,0}^i}(0\|j\|g) + F_{k_{b,0}^i}(0\|j\|g) \right) + k_{c,\chi_1}^j \\ B_g^j &= \left( \sum_{i=1}^n F_{k_{a,0}^i}(1\|j\|g) + F_{k_{b,1}^i}(0\|j\|g) \right) + k_{c,\chi_2}^j \\ C_g^j &= \left( \sum_{i=1}^n F_{k_{a,1}^i}(0\|j\|g) + F_{k_{b,0}^i}(1\|j\|g) \right) + k_{c,\chi_3}^j \\ D_g^j &= \left( \sum_{i=1}^n F_{k_{a,1}^i}(1\|j\|g) + F_{k_{b,1}^i}(1\|j\|g) \right) + k_{c,\chi_4}^j \end{aligned}$$

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

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

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

**Pre-Processing:** This procedure is performed as follows

1. Call **Initialize** on  $\mathcal{F}_{\text{offline}}$  with the smallest prime  $p$  in  $\{2^\kappa, \dots, 2^{\kappa+1}\}$ .
2. Call **Preprocessing-I** on  $\mathcal{F}_{\text{offline}}$  with input  $W$  and  $G$ .
3. Call **Preprocessing-II** on  $\mathcal{F}_{\text{offline}}$  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  $\rho_w$  and computes  $\Lambda_w = \rho_w \oplus \lambda_w$ , where  $\lambda_w$  was obtained in the preprocessing stage. The value  $\Lambda_w$  is broadcast to all parties.
2. Party  $i$  calls **Output** on  $\mathcal{F}_{\text{offline}}$  to open  $[k_{w, \Lambda_w}^i]$  for all his input wires  $w$ , we denote the resulting value by  $k_w^i$ .
3. The parties call **Output** on  $\mathcal{F}_{\text{offline}}$  to open  $[\mathbf{A}_g]$ ,  $[\mathbf{B}_g]$ ,  $[\mathbf{C}_g]$  and  $[\mathbf{D}_g]$  for every gate  $g$ .
4. Passing through the circuit topologically, the parties can now locally compute the following operations for each gate  $g$ 
  - Let the gates input wires be labeled  $a$  and  $b$ , and the output wire be labeled  $c$ .
  - For  $j = 1, \dots, n$  compute  $k_c^j$  according to the following cases:

- *Case 1* –  $(\Lambda_a, \Lambda_b) = (0, 0)$ : compute

$$k_c^j = A_g^j - \left( \sum_{i=1}^n F_{k_a^i}(0||j||g) + F_{k_b^i}(0||j||g) \right).$$

- *Case 2* –  $(\Lambda_a, \Lambda_b) = (0, 1)$ : compute

$$k_c^j = B_g^j - \left( \sum_{i=1}^n F_{k_a^i}(1||j||g) + F_{k_b^i}(0||j||g) \right).$$

- *Case 3* –  $(\Lambda_a, \Lambda_b) = (1, 0)$ : compute

$$k_c^j = C_g^j - \left( \sum_{i=1}^n F_{k_a^i}(0||j||g) + F_{k_b^i}(1||j||g) \right).$$

- *Case 4* –  $(\Lambda_a, \Lambda_b) = (1, 1)$ : compute

$$k_c^j = D_g^j - \left( \sum_{i=1}^n F_{k_a^i}(1||j||g) + F_{k_b^i}(1||j||g) \right).$$

- If  $k_c^i \notin \{k_{c,0}^i, k_{c,1}^i\}$ , 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  $k_c^i = k_{c,0}^i$  then  $P_i$  sets  $\Lambda_c = 0$ ; if  $k_c^i = k_{c,1}^i$  then  $P_i$  sets  $\Lambda_c = 1$ .
- The output of the gate is defined to be  $(k_c^1, \dots, k_c^n)$  and  $\Lambda_c$ .

5. Assuming party  $P_i$  does not abort it will obtain  $\Lambda_w$  for every circuit-output wire  $w$ . The party can then recover the actual output value from  $\rho_w = \Lambda_w \oplus \lambda_w$ , where  $\lambda_w$  was obtained in the preprocessing stage.

enters more than one gate) need to be identical in all of these gates. In addition, the pseudorandom function values may not be correctly computed from the pseudorandom function keys that are input. In order to make the computation of the garbled circuit efficient, we will not check that the pseudorandom function values are correct. However, it is necessary to ensure that the  $\lambda_w$  values are correct, and that they (and likewise the keys) are consistent between gates (e.g., as in the case where the same wire is input to multiple gates). We achieve this by computing the entire circuit at once, via a single functionality.

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

Another issue that arises is that the single garbled gate functionality expects to receive a single masking value for each wire. However, since this value is secret, it must be generated from shares that are input by the parties. In the full version of the paper [15] we describe the full protocol for securely computing  $\mathcal{F}_{\text{offline}}$  in the  $\mathcal{F}_{\text{MPC}}$ -hybrid model (i.e., using *any* protocol that securely computes the  $\mathcal{F}_{\text{MPC}}$  ideal functionality). In short, the parties input shares of  $\lambda_w$  to the functionality, the single masking value is computed from these shares, and then input to all the necessary gates.

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

This issue is solved in the following way. The computation is performed by having the parties input random masking values  $\lambda_w^i \in \{1, -1\}$ , instead of bits. This enables the computation of a value  $\mu_w$  to be the *product* of  $\lambda_w^1, \dots, \lambda_w^n$  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  $\lambda_w^i$  values that are not in  $\{-1, +1\}$ , the protocol for computing the circuit outputs  $(\prod_{i=1}^n \lambda_w^i)^2 - 1$ , for every wire  $w$  (where  $\lambda_w^i$  is the share contributed from party  $i$  for wire  $w$ ), and the parties can simply check whether it is equal to zero or not. Thus, if any party cheats by causing some  $\lambda_w \notin \{-1, +1\}$ , then this will be discovered since the circuit outputs a non-zero value for  $(\prod_{i=1}^n \lambda_w^i)^2 - 1$ , 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  $\lambda_w$  itself.

In the next section we shall remove *all* of the complications by basing our implementation for  $\mathcal{F}_{\text{MPC}}$  upon the specific SPDZ protocol. The reason why the

SPDZ implementation is simpler – and more efficient – is that SPDZ provides generation of such shared values effectively for free.

### 3 The SPDZ Based Instantiation

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

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

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

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

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

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

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

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

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

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

#### 3.1 Utilizing the SPDZ Protocol

As discussed in Section 2.2, 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  $\Pi_{\text{offline}}$  which

implements the functionality  $\mathcal{F}_{\text{offline}}$  by utilizing the specific SPDZ MPC protocol as the underlying implementation of  $\mathcal{F}_{\text{MPC}}$ . These optimizations are possible because the SPDZ MPC protocol provides a richer interface to the protocol designer than the naive generic MPC interface given in functionality  $\mathcal{F}_{\text{MPC}}$ . In particular, it provides the capability of directly generating shared random bits and strings. These are used for generating the masking values and pseudorandom function keys. Note that one of the most expensive steps in FairplayMP [2] was coin tossing to generate the masking values; by utilizing the specific properties of SPDZ this is achieved essentially for free.

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

Throughout, we utilize  $\mathcal{F}_{\text{SPDZ}}$  (Functionality 4), which represents an idealized representation of the SPDZ protocol, akin to the functionality  $\mathcal{F}_{\text{MPC}}$  from Section 2.2. Note that in the real protocol,  $\mathcal{F}_{\text{SPDZ}}$  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  $\mathcal{F}_{\text{SPDZ}}$ . In the SPDZ offline phase we need to know the maximum number of multiplications, random values and random bits required in the online phase. In that phase the random shared bits and values are produced, as well as the “Beaver Triples” for use in the multiplication gates performed in the SPDZ online phase. In particular the consuming of shared random bits and values results in no cost during the SPDZ online phase, with all consumption costs being performed in the SPDZ offline phase. The protocol, which utilizes Somewhat Homomorphic Encryption to produce the shared random values/bits and the Beaver multiplication triples, is given in [7].

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

### 3.2 The $\mathcal{F}_{\text{offline}}$ SPDZ based Protocol

As remarked earlier  $\mathcal{F}_{\text{offline}}$  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, it utilizes the fact that SPDZ can very efficiently generate coin tosses. This means that it is not necessary for the parties to input the  $\lambda_w^i$  values, to multiply them together to obtain  $\lambda_w$  and to output the check values  $(\lambda_w)^2 - 1$ . Thus, this yields a significant efficiency improvement. We now describe the protocol which implements  $\mathcal{F}_{\text{offline}}$  in the  $\mathcal{F}_{\text{SPDZ}}$ -hybrid model

#### preprocessing-I:

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

$$M = 13 \cdot G, \quad B = W, \quad R = 2 \cdot W \cdot n, \quad I = 2 \cdot G \cdot n + W,$$

where  $G$  is the number of gates,  $n$  is the number of parties and  $W$  is the number of input wires per party. In practice the term  $W$  in the calculation of  $I$  needs only be an upper bound on the total number of input wires per party in the circuit which will eventually be evaluated.

2. **Generate wire masks:** For every circuit wire  $w$  we need to generate a sharing of the (secret) masking-values  $\lambda_w$ . Thus for *all* wires  $w$  the parties execute the command **RandomBit** on the functionality  $\mathcal{F}_{\text{SPDZ}}$ , the output is denoted by  $[\lambda_w]$ . The functionality  $\mathcal{F}_{\text{SPDZ}}$  guarantees that  $\lambda_w \in \{0, 1\}$ .
3. **Generate keys:** For every wire  $w$ , each party  $i \in [1, \dots, n]$  and for  $j \in \{0, 1\}$ , the parties call **Random** on the functionality  $\mathcal{F}_{\text{SPDZ}}$  to obtain output  $[k_{w,j}^i]$ . The parties then call **Output** to open  $[k_{w,j}^i]$  to party  $i$  for all  $j$  and  $w$ . The vector of shares  $[k_{w,j}^i]_{i=1}^n$  we shall denote by  $[\mathbf{k}_{w,j}]$ .

**preprocessing-II:** (This protocol implements the computation gate table as it is detailed in the BMR protocol. The correctness of this construction is explained in the full version of the paper.)

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

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

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 can be computed using **Add** and **Multiply**.

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

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

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

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

$$\begin{aligned} [F_{k_{w,b}^i}^0(g)] &= [F_{k_{w,b}^i}^i(0 \| 1 \| g), \dots, F_{k_{w,b}^i}^i(0 \| n \| g)] \\ [F_{k_{w,b}^i}^1(g)] &= [F_{k_{w,b}^i}^i(1 \| 1 \| g), \dots, F_{k_{w,b}^i}^i(1 \| n \| g)]. \end{aligned}$$

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

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

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

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

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

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

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

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

### 3.3 Circuit Complexity

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

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

- **Multiplications per gate.** We will follow the calculation that is done per gate chronologically as it occurs in step 3 of **preprocessing-II** phase:
  1. In order to calculate the indicators in step 3a it suffices to compute one multiplication and 4 squares. We can do this by altering the equations a little. For example, for  $f_g = AND$ , we calculate the indicators by first computing  $[t] = [\lambda_a] \cdot [\lambda_b]$  (this is the only multiplication) and then  $[x_a] = ([t] - [\lambda_c])^2$ ,  $[x_b] = ([\lambda_a] - [t] - [\lambda_c])^2$ ,  $[x_c] = ([\lambda_b] - [t] - [\lambda_c])^2$ , and  $[x_d] = (1 - [\lambda_a] - [\lambda_b] + [t] - [\lambda_c])^2$ .  
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  $[x_d] = [x_a]$ .  
Observe that in XOR gates only two squaring operations are needed.
  2. To obtain the correct vector (in step 3b) which is used in each garbled label, we carry out 8 multiplications. Note that in XOR gates only 4 multiplications are needed, because  $\mathbf{k}_{c,x_c} = \mathbf{k}_{c,x_b}$  and  $\mathbf{k}_{c,x_d} = \mathbf{k}_{c,x_a}$ .  
Summing up, we have 4 squaring operations in addition to 9 multiplication operations per AND gate and 2 squarings in addition to 5 multiplications per XOR gate.
- **Multiplications in the entire circuit.** Denote the number of multiplication operation per gate (i.e. 13 for AND and 7 for XOR) by  $c$ , we get  $G \cdot c$  multiplications for garbling all gates (where  $G$  is the number of gates in the boolean circuit computing the functionality  $f$ ). Besides garbling the gates we have no other multiplication operations in the circuit. Thus we require  $c \cdot G$  multiplications in total.

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

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

**Arithmetic vs Boolean Circuits:** Our protocol will perform favourably for functions which are reasonably represented as boolean circuit, but the low round complexity may be outweighed by other factors when the function can be expressed much more succinctly using an arithmetic circuit, or other programatic representation as in [12]. In such cases, the performance would need to be tested for the specific function.

### 3.4 Expected Runtimes

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

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

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

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

<sup>4</sup> 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.

To see what this would imply in practice consider the AES circuit described in [18]; which has become the standard benchmarking case for secure computation calculations. The basic AES circuit has around 33,000 gates and a similar number of wires, including the key expansion within the circuit.<sup>5</sup> Assuming the parties share a XOR sharing of the AES key, (which adds an additional  $2 \cdot n \cdot 128$  gates and wires to the circuit), the parameters for the **Initialize** call to the  $\mathcal{F}_{\text{SPDZ}}$  functionality in the preprocessing-I protocol will be

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

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

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

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

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

- In [6] an actively secure computation of AES using SPDZ took an offline time of over five minutes per AES block, with an online time of around a quarter of a second; that computation used a security parameter of 64 as opposed to our estimates using a security parameter of 128.
- In [12] another experiment was shown which can achieve a latency of 50 milliseconds in the online phase for AES (but no offline times are given).

---

<sup>5</sup> Note that unlike [18] and other Yao based techniques we cannot process XOR gates for free. On the other hand we are not restricted to only two parties.

- In [17] the authors report on a two-party MPC evaluation of the AES circuit using the Tiny-OT protocol; they obtain for 80 bits of security an amortized offline time of nearly three seconds per AES block, and an amortized online time of 30 milliseconds; but the reported non-amortized latency is much worse. Furthermore, this implementation is limited to the case of *two parties*, whereas we obtain security for multiple parties.

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

## Acknowledgments

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

## References

1. D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In H. Ortiz, editor, *22nd STOC*, pages 503–513. ACM, 1990.
2. A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multiparty computation. In P. Ning, P. F. Syverson, and S. Jha, editors, *ACM CCS*, pages 257–266. ACM, 2008.
3. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In Simon [21], pages 1–10.
4. D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In Simon [21], pages 11–19.

5. S. G. Choi, J. Katz, A. J. Malozemoff, and V. Zikas. Efficient three-party computation from cut-and-choose. In Garay and Gennaro [9], pages 513–530.
6. I. Damgård, M. Keller, E. Larraia, C. Miles, and N. P. Smart. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In I. Visconti and R. D. Prisco, editors, *SCN 2012*, volume 7485 of *LNCS*, pages 241–263. Springer, 2012.
7. I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In J. Crampton, S. Jajodia, and K. Mayes, editors, *ESORICS*, volume 8134 of *LNCS*, pages 1–18. Springer, 2013.
8. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [20], pages 643–662.
9. J. A. Garay and R. Gennaro, editors. *Advances in Cryptology - CRYPTO 2014*, volume 8617 of *LNCS*. Springer, 2014.
10. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In A. V. Aho, editor, *Proceedings STOC 1987*, pages 218–229. ACM, 1987.
11. V. Goyal. Constant round non-malleable protocols using one way functions. In L. Fortnow and S. P. Vadhan, editors, *Proceedings STOC 2011*, pages 695–704. ACM, 2011.
12. M. Keller, P. Scholl, and N. P. Smart. An architecture for practical actively secure MPC with dishonest majority. In A. Sadeghi, V. D. Gligor, and M. Yung, editors, *2013 ACM CCS '13*, pages 549–560. ACM, 2013.
13. E. Larraia, E. Orsini, and N. P. Smart. Dishonest majority multi-party computation for binary circuits. In Garay and Gennaro [9], pages 495–512.
14. Y. Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology - CRYPTO 2013*, volume 8043 of *LNCS*, pages 1–17. Springer, 2013.
15. Y. Lindell, B. Pinkas, N. P. Smart, and A. Yanai. Efficient constant round multi-party computation combining BMR and SPDZ (Full Version). *IACR Cryptology ePrint Archive*, 2015:523, 2015.
16. Y. Lindell and B. Riva. Cut-and-choose yao-based secure computation in the online/offline and batch settings. In Garay and Gennaro [9], pages 476–494.
17. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [20], pages 681–700.
18. B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In M. Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.
19. T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In D. S. Johnson, editor, *Proceedings STOC 1989*, pages 73–85. ACM, 1989.
20. R. Safavi-Naini and R. Canetti, editors. *Advances in Cryptology - CRYPTO 2012*, volume 7417 of *LNCS*. Springer, 2012.
21. J. Simon, editor. *Proceedings STOC 1988*. ACM, 1988.
22. A. C. Yao. Protocols for secure computations. In *Proceedings FOCS 1982*, pages 160–164. IEEE Computer Society, 1982.