BAR-ILAN UNIVERSITY

Concretely Efficient (Constant Round) Protocol for General Secure Multiparty Computation With an Active Adversary

Avishay Yanay

Submitted in partial fulfillment of the requirements for the Master's Degree in the Department of Computer Science, Bar-Ilan University

Ramat-Gan, Israel 2015

BAR-ILAN UNIVERSITY

Concretely Efficient (Constant Round) Protocol for General Secure Multiparty Computation With an Active Adversary

Avishay Yanay

Submitted in partial fulfillment of the requirements for the Master's Degree in the Department of Computer Science, Bar-Ilan University

Ramat-Gan, Israel 2015

This work was carried out under the supervision of

Professor Yehuda Lindell and **Professor Benny Pinkas**

Department of Computer Science, Bar-Ilan

Efficient Constant Round Multi-Party Computation Combining BMR and SPDZ

Avishay Yanay

July 15, 2015

Contents

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 thesis, 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.

Acknowledgments

I would like to thank my supervisors, **Prof. Yehuda Lindell** and **Prof. Benny Pinkas**, for the patient guidance, encouragement and advice they have provided throughout my time as their student. I have been extremely lucky to have supervisors who cared so much about my work, and who responded to my questions and queries so promptly.

Chapter 1

Introduction

1.1 General Background

In secure multiparty computation (MPC) a set of n parties wishes to distributively and securely compute a joint functionality of their inputs $f : (\{0,1\}^{\ell})^n \to (\{0,1\}^m)^n$, that is, party P_i inputs x_i to the the functionality and receives back y_i (where $1 \leq i \leq n$, $|x_i| = \ell$, $|y_i| = m$). A secure protocol π that allows the parties to compute f assumes that some of the parties are distrustful, for instance, a distrustful player P_j tries to learn more than y_j implicitly reveals.
A secure protocol must guarantee several properties:

- *Privacy,* meaning that the parties learn only their output and what implicitly could be learned from it, but nothing else.
- *Correctness,* meaning that the output that the parties receive is correctly computed from their inputs.
- *Independence of inputs,* meaning that the input that is chosen by each party is independent of the other parties' inputs.

In some settings a protocol may guarantee more properties such as

- *Fairness,* meaning that whenever the dishonest parties receive their outputs then the honest parties receive their outputs too.

- *Guaranteed output delivery,* meaning that the honest parties receive their outputs regardless of the dishonest parties' behavior.

Through this methodology (i.e. definition by properties) we assume to know what strategy the adversary chooses (i.e. what properties it would try to break), thus it is mandatory that we do not define the security by a set of properties but rather give a thorough definition that captures even strategies that are not on our mind at the moment. To this end, the security of a protocol is formally defined by comparing the distribution of the outputs of all parties in the execution of the protocol π to an *ideal* model where a trusted third party is given the inputs from the parties, compute f and return the outputs. The idea is that if it is possible to simulate the adversary's view from the real execution of the protocol in the ideal model (when it only sees its input and output), then it follows that the adversary cannot do in the real execution anything that is impossible in the ideal model, and hence the protocol is said to be secure.

In MPC we usually consider the capability of the adversary, i.e. what the adversary is allowed (or is able) to do in order to harm the security of the protocol; the main types of adversaries are *semi-honest* (also called *honest but curious*) who follow the protocol specification but tries to learn more than allowed by inspecting the transcript, and *malicious* who attempts to deviate from the specified protocol in order to break the security (e.g. send modified or new messages that have not been specified by the protocol). There are other properties associated with adversaries such as *static* adversary - who corrupts a set of parties before the execution of the protocol begins; *adaptive* adversary - who may corrupt different parties during the execution of the protocol; *honest majority* refers to an adversary who may corrupt less than half of the parties and *dishonest majority* refers to an adversary who can corrupt arbitrary number of parties (it is obvious that a secure two-party protocol is secure in the model of dishonest majority). It is worth to note that we consider the corrupted parties as if they are controlled by a single external adversary and thus their behaviour during the protocol could be coordinated to achieve the best.

Malicious behaviour

There are several types of malicious behaviours that are unavoidable even in the ideal model: the adversary might corrupt a party so it does not input its original input value x to the protocol but rather uses a modified input $x' \neq x$; it could instruct the corrupted party to output a value y' that is different from the output y given from the trusted party (that value y' might be the output of any probabilistic polynomial time algorithm operating on the adversary's random tape, set of inputs and all the messages it have seen so far); last, the adversary might instruct a corrupted party to *abort* (e.g. shut himself down) before or after any step of the protocol (even before the protocol has begun). In a real execution, this last type of behaviour might cause all parties to abort, or otherwise, there is some recovery mechanism that allows the other parties to keep computing the functionality. It had been studied ([Cle86]) that in a dishonest majority model, if the adversary wish to abort the protocol, it implies that there exist no recovery mechanism (still, there could be a mechanism that identifies the corrupted party that caused the premature abortion, this is called *identified abortion*).

Initial Solutions

The first solution to the two-party problem was introduced by Yao [Yao82] and was proved to be secure for the semi-honest model only, however, Yao's protocol was extended to the malicious model by Lindell and Pinkas [LP07] via the "Cut and Choose" method. Goldreich, Micali and Wigderson [GMW87] presented protocols for both the two-party and multi-party cases which are secure in the semi-honest model, and a method to "compile" a protocol such that the resulted protocol would be secure even in the malicious model.

These general solutions prove that secure multi-party computation is feasible. However, for many years, both protocols were considered to be inefficient - in particular, for the malicious adversary case - and thus far from being used in practice. As secure computation moves from theory to practice, there are two approaches to deal with the inefficiency problem. The first approach attempts to tailor solutions to specific problems instead of the general solutions. The second approach seeks ways to improve the general solution and reduce the gap between the protocols

performance and the real world.

Parameters

Among the parameters by which we analyze a secure computation protocol there are the *computational complexity* which, as usual, considers the amount of instructions that the parties have to execute in order to achieve their outputs; *round complexity* which considers the number of communication rounds, where a communication round is a state in the protocol in which the party sends/receives one message; and *message complexity* which consider the amount of information that is sent during the protocol execution in order to accomplish the computation. In practice, the overhead time wasted for opening/closing a communication channel and sending a message (especially in a wide area networks) is relatively large, hence, we are interested in protocols that have a minimum number of communication rounds. Although the fastest protocols today [DPSZ12, NNOB12] are fast in terms of computational complexity, they have a bottleneck when the depth of the circuit is large because they require the parties to communicate for every multiplication gate in the circuit, which makes the round complexity equal to the circuit's depth.

Performance Improvements

Bar-Ilan and Beaver [BB89] were the first to investigate reducing the round complexity for secure function evaluation. They exhibited a non-cryptographic method that always saves a logarithmic factor of rounds (logarithmic in the total length of the players' inputs), while the message complexity grows only by a polynomial factor. Alternatively, they showed that the number of rounds can be reduced to a constant, but at the expense of an exponential blowup in the message sizes. Beaver, Micali and Rogaway [BMR90] proved that it is possible to achieve a constant round protocol while preserving the polynomial message complexity. Their protocol is considered as a direct generalization of Yao's protocol with respect to the way the parties distributively garble the circuit.

In the following we overview few of the classic works, including Yao's protocol for the two-party case, the GMW approach for both the two-party and the multi-party cases and the BMR

constant round protocol in the multi-party case, along with an overview of the current state of the art solutions to the multi-party and dishonest majority case ([DPSZ12]).

1.2 Work Structure

We review the most known and related works in the field in section 1.3 and summarize our improvements in section 1.4.

In chapter 2 we present our modification to the oroginal BMR protocol and present our general protocol, in the $\mathcal{F}_{\mathsf{offline}}$-hybrid model, that can utilize any MPC protocol for arithmetic circuits as a subprocedure.

We then present our specialization of the previous protocol to the case of utilizing the SPDZ [DPSZ12] protocol as the underlying MPC protocol in chapter 3. This enables further optimizations which are not available in a generic MPC protocol, and thus enables an even more efficient evaluation.

In chapter 4 we give a full proof of security, which is achieved by a reduction to the original BMR protocol.

1.3 Previous Works

1.3.1 Yao's two-party protocol ([Yao82])

Suppose that two parties P_1 and P_2, having private inputs x and y wish to obtain the value of a known two-argument function evaluated on them, that is, we consider functionalities of the form $(x, y) \mapsto (f(x, y), f(x, y))$ (this does not necessarily mean that both parties learn the same output, it might be that the output is composed of two encrypted values where each party can decrypt only one of them). Further suppose that the parties agreed on and hold a boolean circuit that computes the functionality f. Note that in Yao's solution the roles of the two parties are not symmetric, i.e. they have different set of instructions to follow, also, recall that this protocol is secure only in the semi-honest model. The idea is that P_1 creates a *garbled* form of the circuit such that P_2 can propagate encrypted values through it and obtain the output in the clear, while

all intermediate values remain secret. The protocol is composed of the following phases:

1. **Garbling the circuit.** This is done by P_1. For every wire w in the circuit two random keys k_w^0 and k_w^1 are chosen, where k_w^0 corresponds to the value 0 passing through that wire and k_w^1 corresponds to the value 1 passing through that wire. The idea is that when P_2 obtains one of the keys then it cannot tell whether it corresponds to the value 0 or 1 since both the keys are random and taken from the same distribution.

   Then, P_1 separately garbles each gate: let $g : \{0,1\} \times \{0,1\} \to \{0,1\}$ be the boolean function of gate g, wires w_1 and w_2 be the input wires to the gate and w_3 be the output wire of the gate (i.e. k_1^0 and k_1^1 are associated with wire w_1 and so on). P_1 computes 4 ciphertexts such that each one of them corresponds to one possibility of input pair for wires (w_1, w_2), each ciphertext is computed by double encrypting the appropriate key of wire w_3 using the keys that correspond to the input pair of (w_1, w_2). For instance, for input pair $(0,1)$ the key of wire w_3 that is encrypted is $k_3^{g(0,1)}$ and the keys used to encrypt it are k_1^0 and k_1^1 (i.e. the 0-key for wire 0 and 1-key for wire 1). The ciphertext that are computed for gate g are:

$$c_{0,0} = E_{k_1^0}\big(E_{k_2^0}(k_3^{g(0,0)})\big) \qquad c_{0,1} = E_{k_1^0}\big(E_{k_2^1}(k_3^{g(0,1)})\big)$$
$$c_{1,0} = E_{k_1^1}\big(E_{k_2^0}(k_3^{g(1,0)})\big) \qquad c_{1,1} = E_{k_1^1}\big(E_{k_2^1}(k_3^{g(1,1)})\big)$$

   where E is the encryption algorithm from an encryption scheme (G, E, D) that is indistinguishable for multiple encryptions, moreover, the scheme should have *elusive efficiently verifiable range* (details in [LP09]), meaning that the party that decrypt can easily determine whether the given value is a legit ciphertext computed using a given key. This way, if P_2 holds the key that corresponds to value b_1 for wire w_1 (i.e. $k_{w_1}^{b_1}$) and the key $k_{w_2}^{b_2}$ for wire w_2, then, using the table above it can first tell which entry from the table is a legit ciphertext computed from the keys $k_{w_1}^{b_1}, k_{w_2}^{b_2}$ and then, using that entry, obtain the key that corresponds to the value $g(b_1, b_2)$, i.e. $k_3^{g(b_1,b_2)}$, without revealing any of the other three values. After computing the four entries, P_1 randomly shuffles them before handing it to P_2 so P_2 wouldn't be able to conclude what are the values of the input wires from the position

of the legit ciphertext.

2. **Sending the garbled circuit.** The first player P_1 provides P_2 with the followings:

   - *Garbled circuit.* That it the set of all garbled gates (i.e. 4-entry tables) that computed before.
   - *Input keys.* The keys that corresponds to the input bits of the input of P_1, for instance, if wire w is an input wire of P_1 and the input bit that it wants to evaluate the circuit with is b then P_1 sends P_2 the key k_w^b, otherwise, it sends k_w^{b-1}.
   - *Translation of output wires.* To allow P_2 to evaluate the circuit and achieve the output in the clear, it needs a translation from keys to bits, that is, for every circuit-output wire w, P_1 sends the ordered pairs $(k_w^0, 0)$ and $(k_w^1, 1)$.

3. **Oblivious transfer.** In order to evaluate the circuit, P_2 has to obtain the keys that correspond to its input as well (in the previous step it obtained the keys that correspond to P_1's inputs), then, P_2 asks those keys from P_1. Obviously P_1 must not learn what keys P_2 asks and the transfer should not be executed in a simple manner, this is achieved using a 1-out-of-2 Oblivious Transfer. That is, for every circuit-input wire w that belongs to P_2 the parties securely computes the functionality $((k_w^0, k_w^1), b) \mapsto (\lambda, k_w^b)$ where $b \in \{0, 1\}$ and λ is the empty string. That is P_2 enters the index of the keys that it needs and eventually obtains that key while P_1 enters both keys and learn nothing from the execution.

4. **Locally evaluating the garbled circuit.** The party P_2 evaluate the circuit gate-by-gate, starting from the circuit-input, for which it knows one key per wire, toward the circuit output, for which it knows the translation from keys to actual bits, thus P_2 obtains the output value of the circuit, and sends it to P_1.

1.3.2 The GMW methodology ([GMW87]

In contrast to Yao's solution (that is based on boolean circuits), the GMW approach is based on arithmetic circuits. The protocol begins in a step where each party obtains a share of the value that is associated with each circuit-input wires, i.e. for the secret value s associated with wire a

the parties $P_1, \ldots, P_n$ obtain the shares $a_1, \ldots, a_n$ where $s_a = a_1 + \ldots + a_n$. Recall that arithmetic circuits are composed of addition and multiplication gates (the operations are done over $\mathbb{F}_2$). Suppose that the parties hold shares for the values associated with wires a and b that enter to an addition gate g (i.e. the secrets s_a and s_b) and want to obtain shares to the secret value associated with g's output wire c (i.e. they want to obtain shares to s_c), then each party P_i only needs to add its own shares $a_i + b_i$ and the result is a share to s_c. That is, if $s_a = a_1 + \ldots + a_n$ and $s_b = b_1 + \ldots + b_n$ then $s_c = s_a + s_b = a_1 + \ldots + a_n + b_1 + \ldots + b_n$. Obtaining shares to an output wire of a *multiplication* gate is more challenging though. First consider the two-party case and later the multiparty case:

Two-party case. Parties P_1 and P_2 holds the shares (a_1, b_1) and (a_2, b_2) respectively, such that $a = a_1 + a_2$, $b = b_1 + b_2$ and wants to obtain shares (c_1, c_2) such that $a \cdot b = c = c_1 + c_2$. This is done using a 1-out-of-4 Oblivious Transfer in the following manner:

- Party P_i holds $(a_i, b_i) \in \{0,1\} \times \{0,1\}$, for $i = 1, 2$.
- Party P_1 uniformly selects $c_1 \in \{0,1\}$
- The parties compute the functionality $((a_1, b_1, c_1), (a_2, b_2)) \mapsto (\lambda, f_{a_2,b_2}(a_1, b_1, c_1))$ where $f_{a,b}(x, y, z) = z + (x + a) \cdot (y + b)$. They privately compute that functionality by a 1 out of 4 OT such that P_1 is the sender and sets its input to be

$$\big(f_{0,0}(a_1, b_1, c_1)\ ,\ (f_{0,1}(a_1, b_1, c_1))\ ,\ f_{1,0}(a_1, b_1, c_1)\ ,\ f_{1,1}(a_1, b_1, c_1)\big)$$

  and P_2 is the receiver and sets its input to be $1 + 2a_2 + b_2 \in \{1, 2, 3, 4\}$. It is easy to see that the computation is correct:

P_2's input, i.e. (a_2, b_2)	Receiver's inputs in OT_1^4	Receiver's output in OT_1^4
$(0,0)$	1	$c_1 + a_1 b_1$
$(0,1)$	2	$c_1 + a_1 \cdot (b_1 + 1)$
$(1,0)$	3	$c_1 + (a_1 + 1) \cdot b_1$
$(1,1)$	4	$c_1 + (a_1 + 1) \cdot (b_1 + 1)$

- Party P_1 outputs c_1 whereas Party P_2 outputs the result obtained from the OT_1^4 execution.

The security of the entire protocol (i.e. distributed evaluation of an arithmetic circuit over a finite field $\mathbb{F}_2$) is reduced to the security of the OT_4^1 protcol, which is based on the existence of family of enhanced trapdoor permutations. Note that the initial distribution of the shares is done in the obvious manner, i.e. party that holds the secret $s \in \{0,1\}$ shares it by choosing random bit from $0,1$, denoted by s_2, and hands it to the other party, then the party sets its own share to be $s_1 = s - s_2$.

Multiparty case. Here we again deal with the multiplication problem, but in contrast to the two-party case, here there is no trivial security reduction to the OT_4^1. Consider the case where the parties $P_1, \ldots, P_n$ holds shares to the secrets a and b, that is, player P_i holds (a_i, b_i) such that $a = \sum_{i=1}^n a_i$ and $b = \sum_{i=1}^n b_i$. The players wish to obtain shares to the secret c such that $c = \sum_{i=0}^n c_i = a \cdot b = (\sum_{i=1}^n a_i) \cdot (\sum_{i=1}^n b_i)$, that is, player P_i obtains c_i. GMW showed a protocol (distributed evaluation of an arithmetic circuit over $\mathbb{F}_2$ in the multiparty case) which its security can be reduced to the security of a the two-party case (which in turn is based on the existence of a family of enhanced trapdoor permutations). The idea is as follows:

$$\begin{aligned}
(\sum_{i=1}^m a_i) \cdot (\sum_{i=1}^m b_i) &= \sum_{i=1}^m a_i b_i + \sum_{1 \le i \le j \le m} (a_i b_j + a_j b_i) \\
&= (2-m) \cdot \sum_{i=1}^m a_i b_i + \sum_{1 \le i \le j \le m} (a_i + a_j) \cdot (b_i + b_j) \\
&= m \cdot \sum_{i=1}^m a_i b_i + \sum_{1 \le i \le j \le m} (a_i + a_j) \cdot (b_i + b_j)
\end{aligned}$$

where the last equality (i.e. the transition from $(2-m)$ to m) stems from the fact the the computation is over $\mathbb{F}_2$. Note that each player P_i can locally compute the first argument of the last equation, while the computation of the second term requires $m - i$ invocations of the two-party computation described above, one invocation per each player P_j where $j \ge i$. The protocol goes as follows:

- **Inputs.** Party P_i holds $(a_i, b_i) \in \{0,1\} \times \{0,1\}$ for $i = 1, \ldots, m$.
- Each pair of parties P_i and P_j where $i < j$ invoke the two-party functionality described above, party P_i provides the input (a_i, b_i) and receives the value $c_i^{i,j}$ as output while party

P_j provides the input (a_j, b_j) and receives the output $c_j^{i,j}$. From the definition of that functionality it follows that $c_i^{i,j} + c_j^{i,j} = (a_i + a_j) \cdot (b_i + b_j)$.

- Party P_i sets $c_i = ma_ib_i + \sum_{j \neq i} c_i^{i,j}$ (it is obvious that $ma_ib_i = 0$ if m is even and $ma_ib_i = a_ib_i$ otherwise).

- Each party outputs c_i.

Malicious adversary. So far the the descriptions above dealt with semi-honest adversary. The GMW approach uses a *compiler* that is given a multi-party protocol that is secure under a semi-honest adversary and outputs a protocol that is secure under the malicious protocol, the role of the compiler is to wrap up each computation step performed by a party into a step in which the party must be able to prove that it performed the computation correctly. If the proof failed then the other parties knows that the first party has cheated and abort the execution. Note that in this approach, one malicious party might cause early abort of the entire execution of the protocol (as mentioned above, this is unavoidable with dishonest majority protocols for general functionalities; however GMW presented another solution, that we don't discuss here, for a model with honest majority where the malicious adversary cannot cause an early abort, that is, the honest party can emulate the parties that aborted and continue the execution with some default values). Before describing the structure of the *compiled* protocol it is important to enumerate what a malicious party may do (beyond whatever a semi-honest party can do):

1. **Modify inputs.** A malicious party may enter the actual execution with an input different from the one that it originally given. The compiler has to guarantee the *independence of inputs* property mentioned in the introduction, that is, the actual input that the party enters is independent of any of the inputs of the honest parties (it might, however, depend on the inputs of the other malicious parties).

2. **Non-uniform random tape.** A malicious party may enter the actual execution with a random tape that is not uniformly distributed. The compiler must prevent this behaviour, i.e. force the parties to use a uniformly distributed random tape.

3. **Sending unspecified messages.** A malicious party may send messages different from what is required in the specification of the protocol. The compiler has to force the malicious player to compute the next message correctly (using the previous messages that it received and its correct uniformly distributed random tape), while in case that the party indeed cheats the other parties will be notified and abort the execution. (In the general case, this neither guarantees output delivery nor fairness, that is, the malicious party might cause an abort right after learning the output and before the honest parties learned it. There are works, however, that show that it is possible to achieve fairness for some specific functionalities, [GHKL08] for example).

The basic structure of the protocol generated by the compiler is follows:

- **Input commitment.** Each party commits to its input bits, it proves, in addition, that it actually knows the value to which it has committed. It follows that each party commits to a value that is essentially independent of the values committed to by the other parties.

- **Coin tossing.** In this phase each party obtains a uniformly distributed random tape, which will assist in the emulation (transforming) the semi-honest secure protocol into the malicious secure one. While each party obtains its own random tape and a decommitment information, the other parties obtain a commitment to this value. This way, in the protocol emulation phase (below) the party could prove, using an NP-statement, that the computation is done according a honest use of the uniform random tape.

- **Protocol emulation.** The parties use *authenticated-computation* in order to emulate each step of the original protocol (i.e. the protocol that is secure under a semi-honest adversary). The emulation guarantees, using invocations of Zero-Knowledge proofs, that the message sent by one party is indeed the next message that should be sent with regard to the party's input to the protocol, its random tape, and the messages that it received so far (in addition to decommitments information) along with the commitment information that the other parties holds. That information is converted into an NP-statement and it has been shown in [GMW86] that it is possible to prove every NP-statement in Zero-Knowledge.

1.3.3 The BMR constant-round protocol ([BMR90])

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

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

Garbling wire values: For each gate g which calculates the function f_g (where $f_g : \{0,1\} \times \{0,1\} \to \{0,1\}$), the garbled gate of g is computed such that the superseeds associated with the output wire are encrypted (via a simple XOR) using the superseeds associated with the input wires, according to the truth table of f_g. Specifically, a superseed $S_{w,0} = s^1_{w,0} \| \cdots \| s^n_{w,0}$ is used to encrypt a value M of length L by computing $M \bigoplus_{i=1}^n G(s^i_{w,0})$, where G is a pseudo-random generator stretching a seed of length κ to an output of length L. This means that *every* one of the seeds that make up the superseed must be known in order to learn the mask and decrypt.

Masking values: Using random seeds instead of the original 0/1 values does not hide the original value if it is known that the first seed corresponds to 0 and the second seed to 1. Therefore, an unknown random *masking bit*, denoted by λ_w, is assigned to wire w (for $0 \leq w < W$). These masking bits remain unknown to the players during the entire protocol,

thereby preventing them from knowing the *real values* ρ_w that pass through the wires. The values that the players *do* know are called the *external values* Λ_w. An external value is defined to be the exclusive-or of the real value and the masking value; i.e., $\Lambda_w = \rho_w \oplus \lambda_w$. When evaluating the garbled circuit the players only see the external values of the wires, which are random bits that tell nothing about the real values, unless they know the masking values. We remark that each party P_i is given the masking value associated with its input. Thus, it can compute the external value itself (based on its actual input) and can send it to all other parties.

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

The output is the garbled gate of g which comprises of a table of four *ciphertexts*, each of them encrypting either $S_{c,0}$ or $S_{c,1}$. The property of the gate construction is that given one superseed for a and one superseed for b it is possible to to decrypt exactly one ciphertext, and reveal the appropriate superseed for wire c (based on the values on the input wires and the gate type). The inputs and outputs of the process which garbles a single gate follows:

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

Inputs:

1. **Seeds:** $s^1_{a,0}, \ldots, s^n_{a,0}$, $s^1_{a,1}, \ldots, s^n_{a,1}$, $s^1_{b,0}, \ldots, s^n_{b,0}$, $s^1_{b,1}, \ldots, s^n_{b,1}$, $s^1_{c,0}, \ldots, s^n_{c,0}$, $s^1_{c,1}, \ldots, s^n_{c,1}$ where each seed is in $\{0,1\}^\kappa$.

2. **PRG output:** The output of G applied to each of the seeds above, such that the first $n \cdot \kappa$

bits of the output are denoted by G^1 and the other $n \cdot \kappa$ bits by G^2.

3. **Masking bits.** Bits λ_a, λ_b and λ_c.

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

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

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

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

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

The BMR Online Phase: In the online-phase the players only have to obtain one superseed for every circuit-input wire, and then every player can evaluate the circuit on its own, without interaction with the rest of the players. The online-phase is described by the following two steps:

Step 1 – send values:

1. Every player P_i broadcasts the external values on the wires associated with its input.

At the end of this step the players know the external value Λ_w for every circuit-input wire w. (Recall that P_i knows λ_w and so can compute Λ_w based on its input.)

2. Every player P_i broadcasts one seed for each circuit-input wire, namely, the Λ_w-seed. At the end of this step the players know the Λ_w-superseed for every circuit-input wire.

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

Correctness: We explain now why the conditions for masking $S_{c,0}$ and $S_{c,1}$ are correct. The external values Λ_a, Λ_b indicate to the parties which ciphertext to decrypt. Specifically, the parties decrypt A_g if $\Lambda_a = \Lambda_b = 0$, they decrypt B_g if $\Lambda_a = 0$ and $\Lambda_b = 1$, they decrypt C_g if $\Lambda_a = 1$ and $\Lambda_b = 0$, and they decrypt D_g if $\Lambda_a = \Lambda_b = 1$.
We need to show that given S_{a,Λ_a} and S_{b,Λ_b}, the parties obtain S_{c,Λ_c}. Consider the case that $\Lambda_a = \Lambda_b = 0$ (note that $\Lambda_a = 0$ means that $\lambda_a = \rho_a$, and $\Lambda_a = 1$ means that $\lambda_a \neq \rho_a$, where ρ_a is the real wire 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.

1.3.4 The SPDZ protocol

Damgård, Pastro, Smart and Zakarias (written [DPSZ12] and pronounced *speeds*) recently presented a practical solution to the multi-party secure computation problem in the dishonest majority case (with a malicious adversary). Their solution is based on the GMW paradigm, and as a result, requires communication rounds for every multiplication gate in the arithmetic circuit (that computes the functionality). The most notable changes from the GMW protocol are:

- The arithmetic circuit C which computes the functionality f is defined over any finite field

$\mathbb{F}_p$ rather than over $\mathbb{F}_2$ in GMW.

- Multiplication gates require communication rounds but unlike the GMW solution which achieves that by invoking a two-round protocol many times, the SPDZ solution computes multiplication directly using Beaver triples (described below).

- In GMW every party has to prove that the message that it sends is indeed the correct one, and does so using a zero-knowledge scheme to prove some NP-statement. In SPDZ the parties share some global secret MAC key, which authenticates the secret values, and is revealed to the players only at the end of the execution, thus, the effort of preserving security is postponed to the end of the execution, and then it becomes a very easy task.

The SPDZ protocol works in the preprocessing model, i.e. the expensive part of the computation is executed in a preprocessing phase while the lightweight part is executed in the online phase. In the offline (preprocessing) phase the parties neither know the circuit nor their inputs to the functionality which will be computed in the online phase. They only prepare the raw materials that are used later (specifically, they distributively generate the Beaver triples). In the online phase the parties distributively evaluate the circuit in a GMW-manner; that is, locally evaluating the addition gates and invoking communication rounds for evaluating multiplication gates. As in GMW, they start from the circuit-input wires and complete the evaluation in the circuit-output wires in which they reconstruct the secret (to reveal the actual output).

In the following we describe the ideas in the SPDZ solution: first we present how the parties evaluate the arithmetic circuit (i.e. the online phase) given the existence of a trusted dealer and later we describe how to implement that trusted dealer (i.e. the offline phase).

To begin with, assume that each party holds an additive share to a global secret MAC key $\alpha \in \mathbb{F}_p$, that is, player P_i holds α_i where $\alpha = \sum_{i=1}^{n} \alpha_i$; secondly assume that each party holds an additive share to the input of every player, for instance, let x be the input of party P_i to the functionality, then, every party P_j holds the share x_j such that $x = \sum_{j=1}^{n} x_j$. Moreover, let

$\gamma(x) = \alpha \cdot (\delta + x)$ be the MAC on x (for some public constant δ); then every party P_j holds the share $\gamma(x)_i$ such that $\gamma(x) = \sum_{j=1}^{n} \gamma(x)_j = \alpha \cdot (\delta + x)$; every such shared value x is denoted by $[x]$; finally, we assume that the parties have an access to a shared triples $[a]$,$[b]$,$[c]$ such that $a \cdot b = c$. All the aforementioned values are produced using the aid of the trusted dealer.

Online phase. The parties distributively evaluate the circuit. That is, they begin from the circuit-input wires, evaluating the gates one after the other, until they reach the shares of the circuit-output wires. Given a sharing $[x], [y]$ of the input wires of a gate g which computes either the addition or the multiplication operation, we now show how to obtain a sharing $[z]$ of the output wire.

- *Addition.* The sharing $[z] = [x + y]$ is obtained without any interaction, i.e. only local computation is required. That is, assume that player P_i holds $(\delta_x, x_i, \gamma(x)_i)$ and $(\delta_y, y_i, \gamma(y)_i)$, then, P_i computes $(\delta_z, z_i, \gamma(z)_i) = (\delta_x + \delta_y, x_i + y_i, \gamma(x)_i + \gamma(y)_i)$. Correctness follows from

  $$\sum_{i=1}^{n}(\gamma(x)_i + \gamma(y)_i) = \sum_{i=1}^{n}\gamma(x)_i + \sum_{i=1}^{n}\gamma(y)_i = \alpha(x + \delta_x) + \alpha(y + \delta_y) = \alpha(x + y + \delta_z) = \gamma(z)$$

  This ability of easily adding a public value is the reason for the public modifier δ in the definition of the shares.

- *Multiplication.* Here, in order to obtain sharing for $[z] = [x \cdot y]$ the parties have to interact as follows:

  1. Use a Beaver triple, i.e. player P_i holds $(a_i, \gamma(a)_i), (b_i, \gamma(b)_i), (c_i, \gamma(c)_i)$ such that $a \cdot b = c$.
  2. Partially open $[x] - [a] = [x - a]$ so everyone obtains $\varepsilon = x - a$. Note that partially opening the secret $[s]$ means that the parties reveal s_i for $i = 1, \ldots, n$ but not $\gamma(s)_i$ since the latter would reveal the MAC key α, which is not desirable.
  3. Partially open $[y] - [b] = [y - b]$ so everyone obtains $\rho = y - b$.
  4. Locally compute the sharing $[z]$ by $[z] = [c] + \varepsilon \cdot [b] + \rho \cdot [a] + \varepsilon \cdot \rho$.

Note that given a sharing $[s]$ it is easy to obtain a sharing $[s+\eta]$ for some constant η; that is, given $[s] = \big(\delta_s, (s_1, \ldots, s_n), (\gamma(s)_1, \ldots, \gamma(s)_n)\big)$ the sharing for $s+\eta$ is

$$[s+\eta] = \big(\delta_s - \eta, (s_1 + \eta, \ldots, s_n), (\gamma(s)_1, \ldots, \gamma(s)_n)\big)$$

thus, when reconstructing the MAC on s the parties compute
$\frac{\sum_{i=1}^{n} \gamma(s)_i}{\alpha} - (\delta_s - \eta) = s + \delta_s - \delta_s + \eta = s + \eta$

5. Correctness follows from:

$$\begin{aligned} c + \varepsilon \cdot b + \rho \cdot a + \varepsilon \cdot \rho &= ab + (x-a)b + (y-b)a + (x-a)(y-b) \\ &= ab + (xb - ab) + (ya - ab) + (xy - xb - ya + ab) \\ &= xy \end{aligned}$$

Offline phase. The purpose of this phase is to prepare the Beaver's triples $[a], [b], [c]$ which are used in the online phase. This phase assumes an Fully Homomorphic Encryption (FHE) scheme with keys pk, sk where the message space is $\mathbb{F}_p$, so given $c_1 = E_{pk}(m_1)$ and $c_2 = E_{pk}(m_2)$ we have

$$D_{sk}(c_1 + c_2) = m_1 + m_2 \quad \textit{and} \quad D_{sk}(c_1 \cdot c_2) = m_1 \cdot m_2$$

where E and D are the encryption and decryption algorithms respectively. In addition, we require that the scheme enables to share the secret key sk among n parties such that party P_i holds sk_i and together, the parties can decrypt a ciphertext c by $D_{sk_1,\ldots,sk_n}(c)$. Note that it is easy to achieve the encryption of the MAC key α (which is used to obtain the authentication code of some message as mentioned above), that is, the parties only broadcast the encryption of their shares, i.e. $E_{pk}(\alpha_i)$, this way, using the additive homomorphic property each player holds $E_{pk}(\alpha)$. The description of generating Beaver's triple $[a], [b], [c]$ is divided to the following:

- *Resharing a secret.* Given an encryption of a secret $c_s = E_{pk}(s)$, the parties want obtain the sharing $[s]$, i.e. shares s_i of the secret itself (not its encryption) such that $s = \sum s_i$. This is achieved by the following procedure:

1. Party P_i generates a random $f_i \in \mathbb{F}_p$ and transmits $c_{f_i} = E_{pk}(f_i)$
2. All compute $c_{s+f} = c_s + \sum c_{f_i}$.
3. Execute $D_{sk_1,\ldots,sk_n}(c_{s+f})$ to obtain $s + f$.
4. Party P_1 sets $s_1 = (s + f) - f_1$, the rest of the parties P_j $(j \neq 1)$ sets $s_j = (-f_j)$. It follows that $\sum_{i=1}^{n} s_i = s + f - \sum_{i=0}^{n} f_i = s + f - f = s$ as required.

- *Generating* $[a], [b]$. Given a way to *Reshare*, it is possible to generate $[a], [b]$ by the following procedure:

  1. Party P_i generate a random a_i and transmits $c_{a_i} = E_{pk}(a_i)$.
  2. All compute $c_a = \sum c_{a_i}$.
  3. All computes $c_{\alpha a} = c_\alpha \cdot c_a$ (as mentioned above, the players holds c_α.
  4. Execute Reshare on c_a and $c_{\alpha a}$ ($c_{\alpha a}$ is the MAC $\gamma(a)$ with initial $\delta_a = 0$).

  The same is done to compute $[b]$.

- *Generating* $[c]$. Recall that it is required that $c = a \cdot b$. Given c_a, c_b the parties achieve $[c]$ by:

  1. All compute $c_c = c_{ab} = c_a c_b$.
  2. Execute Reshare on c_c so $[c]$ is obtained.

Note that the offline phase is based on a FHE scheme which makes the computation expensive, while in the online phase the parties execute only simple operations. Moreover, given a trusted dealer which implements the offline phase then the resulting protocol is information theoretically secure against up to $n - 1$ maliciously corrupted parties.

1.4 Our Results

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. There are two basic ideas behind our construction:

- We 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.

- More importantly, we observed that once we use such a protocol then the only damage that the adversary can do is to change the reported wire values[1] – but this is easily detectable by the honest parties.

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 [DPSZ12].

1.4.1 Correctness in Offline Phase

In the offline phase of the BMR construction for the *malicious* model the parties have to prove that the PRG results that they provide to the computation of the garbled circuit are computed correctly (this step is not mentioned in Section 1.3.3 since we only described the BMR protocol for the semi-honest case).

A crucial observation, resulting in a great performance improvement, shows that the parties *are not required to verify the correctness* of the computations of the different tables (of the garbled gates). Rather, validation of the correctness is an immediate byproduct of the online computation phase, and therefore does not add any overhead to the computation.

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

[1] This values are the PRF result applied to the keys that are opened to the parties, this is thoroughly described later.

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 [DPSZ12], 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.*

1.4.2 Forcing consistency of inputs

In our general construction of the protocol the parties generate the keys and masking values by themselves (unlike in the construction with SPDZ, in which the keys and masking values are generated by the functionality). This raises an issue in the BMR construction where the parties invoke an independent instantiation of the underlying MPC protocol[2] for every gate, for example: let w be an output wire of gate g and an input wire to gate g', thus, a corrupted party P_i might input to the first invocation of the underlying MPC protocol (computing the garbled gate of g) its share λ_w^i of masking value which leads to a masking value λ_w, and to the second invocation (computing the garbled gate of g') a different share $\overline{\lambda_w^i}$ which leads to a masking value $\overline{\lambda_w}$.
We solve this potential for inconsistency by invoking the underlying functionality (SPDZ) only once (that is, all the gates of the circuit C (which computes f) are garbled using a single big arithmetic circuit), and still preserving the concurrency of the computation of the garbled circuit. This way the parties input their keys to the functionality once and for all, hence, they are not able to input inconsistent keys.
Note that above issue relates only to inconsistency with regard to shares of masking values and not with keys or PRF results.; This is to the fact that for a given gate g with input wires a, b and output wire c, in order to computed the garbled gate of g the parties provide the PRF results for wires a, b and the keys for wire c, hence, they are not required to input the same value twice for some wire.

1.4.3 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 [BNP08] in the case of semi-honest security with an honest majority. In contrast to [BNP08], 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]Specifically, GMW is used in BMR construction.

Chapter 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 described in Section 1.3.3. 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 [BNP08] in the FairplayMP system, which used the BGW protocol [BGW88] 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[1] 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

[1] The external values (as denoted in [BNP08]) are the *signals* (as denoted in [BMR90]) observable by the parties when evaluating the circuit in the online phase.

computations of different gates. This type of verification would greatly increase the cost since the evaluation of the pseudorandom functions (or pseudorandom generator in the original BMR) used in computing the tables needs to be be checked inside the secure computation. This means that the pseudorandom function is not treated as a black box, and the circuit for the offline phase would be huge (as it would include multiple copies of a subcircuit for computing pseudorandom function computations for every wire). Instead, we prove that this type of corrupt behavior can only result in an abort in the online phase, which would not affect the security of the protocol. This observation enables us to compute each gate independently and model the pseudorandom function used in the computation as a black box, thus simplifying the protocol and optimizing its performance.

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

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

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

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

uniform is given by

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

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

Functionality 1 (The SFE Functionality: $\mathcal{F}_{\mathsf{SFE}}$)**.**
The functionality is parameterized by a function $f(x_1,\ldots,x_n)$ which is input as a binary circuit C_f. The protocol consists of three 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: Waits on the adversary to return a *GO/NO-GO* decision. If the adversary returns *NO-GO* then the functionality aborts.

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

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

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

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

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

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

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

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

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

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

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

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

2.1 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 3. In the next section, we present an efficient methodology to implement $\mathcal{F}_{\mathsf{offline}}$ which uses the SPDZ protocol as the underlying MPC protocol for securely computing functionality $\mathcal{F}_{\mathsf{MPC}}$; while in Appendix 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.

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

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

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

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

2.3 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

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

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

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

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

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

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

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

  - Define the selector variables

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

  - 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 $1 \leq j \leq n$:

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

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

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

On input of a circuit C_f representing the function f which consists of at most W wires and at most G gates the parties execute the following commands.
Pre-Processing: This procedure is performed as follows

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

Online Computation: This procedure is performed as follows

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

gate of the circuit (that is, for each gate the parties provide their keysfor the output wire, the PRF results applied to the keys of the input wires and shares of the masking values for all wires associated with that gate). This is sufficient when considering semi-honest adversaries. However, in the setting of malicious adversaries, this can be problematic since parties may input inconsistent values. For example, the masking values λ_w that are common to a number of gates (which happens when any wire enters more than one gate) need to be identical in all of these gates. In addition, the pseudorandom function values may not be correctly computed from the pseudorandom function keys that are input. In order to make the computation of the garbled circuit efficient, we will not check that the pseudorandom function values are correct. However, it is necessary to ensure that the λ_w values are correct, and that they 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 λ_w^i[2] from party P_i *only once*, regardless of the number of gates to which w is input. Thereby consistency is immediate throughout, and this potential attack is prevented. Moreover, the λ_w values are generated once and used consistently by the circuit, making it easy to ensure that the λ values are correct.

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

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

[2]In addition to the values $k_{w,0}^i, k_{w,1}^i$ and the output of F applied to these keys.

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 μ_w to be the *product* of $\lambda_w^1,\ldots,\lambda_w^n$ 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 setting the final masking value λ_w values to be different from ± 1, the protocol for computing the circuit outputs $(\prod_{i=1}^n \lambda_w^i)^2 - 1$, for every wire w (where λ_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 $\lambda_w \notin \pm 1$, 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 $\prod_{i=1}^n \lambda_w^i \in \pm 1$, then the additional value output reveals nothing about λ_w itself.

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

Chapter 3

The SPDZ Based Instantiation

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

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

Initialize: On input $(init, p, M, B, R, I)$ from all parties, the functionality activates and stores p. Preprocessing 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: 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 chapter 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 garbles that binary circuit. In this section we simplify and optimize the implementation of the protocol Π_{offline} which implements the functionality $\mathcal{F}_{\mathsf{offline}}$ by utilizing the specific SPDZ MPC protocol as the underlying implementation of $\mathcal{F}_{\mathsf{MPC}}$. These optimizations are possible because the SPDZ MPC protocol provides a richer interface to the protocol designer than the naive generic MPC interface given in functionality $\mathcal{F}_{\mathsf{MPC}}$. In particular, it provides the capability of directly generating shared random bits and strings. These are used for generating the masking values and pseudorandom function keys. Note that one of the most expensive steps in FairplayMP [BNP08] 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 complexity analysis of the circuit appears in Section 3.3 and the expected results from an implementation of the circuit using the SPDZ protocol are in Section 3.4.

Throughout, we utilize $\mathcal{F}_{\mathsf{SPDZ}}$ (Functionality 4), which represents an idealized representation of the SPDZ protocol, akin to the functionality $\mathcal{F}_{\mathsf{MPC}}$ from chapter 2. Note that in the real protocol, $\mathcal{F}_{\mathsf{SPDZ}}$ is implemented itself by an offline phase (essentially corresponding to our preprocessing-I) and an online phase (corresponding to our preprocessing-II). We fold the SPDZ offline phase into the **Initialize** command of $\mathcal{F}_{\mathsf{SPDZ}}$. In the SPDZ offline phase we need to know the maximum number of multiplications, random values and random bits required in the online phase. In that phase the random shared bits and values are produced, as well as the "Beaver Triples" for use in the multiplication gates performed in the SPDZ online phase. In particular the consuming of shared random bits and values results in no cost during the SPDZ online phase, with all consumption costs being performed in the SPDZ offline phase. The protocol, which utilizes Somewhat Homomorphic Encryption to produce the shared random values/bits and the Beaver multiplication triples, is given in [DKL$^+$13].

As before, we use the notation [*varid*] to represent the result stored in the variable *varid* by the

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

3.2 The Π_{offline} SPDZ based Protocol

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

preprocessing-I:

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

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

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

2. **Generate wire masks:** For every circuit wire w we need to generate a sharing of the (secret) masking-values λ_w. Thus for *all* wires w the parties execute the command **RandomBit** on the functionality $\mathcal{F}_{\mathsf{SPDZ}}$, the output is denoted by $[\lambda_w]$. The functionality $\mathcal{F}_{\mathsf{SPDZ}}$ guarantees that $\lambda_w \in \{0, 1\}$.

3. **Generate keys:** For every wire w, each party $i \in [1, \ldots, n]$ and for $j \in \{0, 1\}$, the parties call **Random** on the functionality $\mathcal{F}_{\mathsf{SPDZ}}$ to obtain output $[k_{w,j}^i]$. The parties then call

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

preprocessing-II: (This protocol implements the computation of the gate table as it is detailed in the BMR protocol. The correctness of this construction is explained at the end of Section 1.3.3.)

1. **Output input wire values:** For all wires w which are attached to party P_i we execute the command **Output** on the functionality $\mathcal{F}_{\mathsf{SPDZ}}$ to open $[\lambda_w]$ to party i.

2. **Output masks for circuit-output-wires:** In order to reveal the real values of the circuit-output-wires it is required to reveal their masking values. That is, for every circuit-output-wire w, the parties execute the command **Output** on the functionality $\mathcal{F}_{\mathsf{SPDZ}}$ for the stored value $[\lambda_w]$.

3. **Calculate garbled gates:** This step is operated for each gate g in the circuit in parallel. Specifically, let g be a gate whose input wires are a, b and output wire is c. Do as follows:

   (a) **Calculate output indicators:** This step calculates four indicators $[x_a]$, $[x_b]$, $[x_c]$, $[x_d]$ whose values will be in $\{0, 1\}$. Each one of the garbled labels $\mathbf{A}_g, \mathbf{B}_g, \mathbf{C}_g, \mathbf{D}_g$ is a vector of n elements that hide either the vector $\mathbf{k}_{c,0} = k^1_{c,0}, \ldots, k^n_{c,0}$ or $\mathbf{k}_{c,1} = k^1_{c,1}, \ldots, k^n_{c,1}$; which one it hides depends on these indicators, i.e if $x_a = 0$ then $\mathbf{A}_g$ hides $\mathbf{k}_{c,0}$ and if $x_a = 1$ then $\mathbf{A}_g$ hides $\mathbf{k}_{c,1}$. Similarly, $\mathbf{B}_g$ depends on x_b, $\mathbf{C}_g$ depends on x_c and $\mathbf{D}_c$ depends on x_d. Each indicator is determined by some function on $[\lambda_a]$, $[\lambda_b]$,$[\lambda_c]$ and the truth table of the gate f_g. Every indicator is calculated slightly differently, 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 $\overset{?}{\neq}$ is defined as $[a] \overset{?}{\neq} [b]$ equals $[0]$ if $a = b$, and equals $[1]$ if $a \neq b$. For the XOR function on a and b, for example, the operator can be evaluated by computing $[a] + [b] - 2 \cdot [a] \cdot [b]$. Thus, these can be computed using **Add** and **Multiply**.

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

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

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

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

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

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

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

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

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

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

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

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

3.3 Circuit Complexity

In this section we analyze the complexity of the above circuit in terms of the number of multiplication gates and of its depth. We are highly concerned with multiplication gates since, given the SPDZ shares $[a]$ and $[b]$ of the secrets a, and b resp., an interaction between the parties is required to achieve a secret sharing of the secret $a \cdot b$. Achieving a secret sharing of a linear combination of a and b (i.e. $\alpha \cdot a + \beta \cdot b$ where α and β are constants), however, can be done

locally and is thus considered negligible. We are interested in the depth of the circuit because it gives a lower bound on the number of rounds of interaction that our circuit requires (note that here, as before, we are concerned with the depth in terms of multiplication gates).

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

- **Multiplications per gate.** We will follow the calculation that is done per gate chronologically as it occurs in Step 3 of preprocessing-II phase:

    1. In order to calculate the indicators in Step 3a it suffices to compute one multiplication and 4 squares. We can do this by altering the equations a little. For example, for $f_g = AND$, we calculate the indicators by first computing $[t] = [\lambda_a] \cdot [\lambda_b]$ (this is the only multiplication) and then $[x_a] = ([t] - [\lambda_c])^2$, $[x_b] = ([\lambda_a] - [t] - [\lambda_c])^2$, $[x_c] = ([\lambda_b] - [t] - [\lambda_c])^2$, and $[x_d] = (1 - [\lambda_a] - [\lambda_b] + [t] - [\lambda_c])^2$.

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

    As another example, for $f_g = XOR$, we first compute $[t] = [\lambda_a] \oplus [\lambda_b] = [\lambda_a] + [\lambda_b] - 2 \cdot [\lambda_a] \cdot [\lambda_b]$ (this is the only multiplication), and then $[x_a] = ([t] - [\lambda_c])^2$, $[x_b] = (1 - [\lambda_a] - [\lambda_b] + 2 \cdot [t] - [\lambda_c])^2$, $[x_c] = [x_b]$, and $[x_d] = [x_a]$.

$$[x_a] = ([t] - [\lambda_c])^2$$

$$[x_b] = (1 - [\lambda_a] - [\lambda_b] + 2 \cdot [t] - [\lambda_c])^2$$

$$[x_c] = [x_b]$$

$$[x_d] = [x_a]$$

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

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

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

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

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

Other Considerations: The overall cost of the pre-processing does not just depend on the number of multiplications. Rather, the parties also need to produce the random data via calls to

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 3.1: SPDZ offline generation times in milliseconds per operation

Random and **RandomBit** to the functionality $\mathcal{F}_{\mathsf{SPDZ}}$.[1] 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 [KSS13]. 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 [DPSZ12, DKL$^+$13]. The offline phase of our protocol runs both the offline and online phases of the SPDZ protocol. The numbers below refer to the SPDZ offline phase, as described in [DKL$^+$13], with covert security and a 20% probability of cheating, using finite fields of size 128-bits, to obtain the following generation times (in milli-seconds). As described in [DKL$^+$13], comparable times are obtainable for running in the fully malicious mode (but more memory is needed).

The implementation of the SPDZ online phase, described in both [DKL$^+$13] and [KSS13], reports online throughputs of between 200,000 and 600,000 per second for multiplication, depending on the system configuration. As remarked earlier the online time of other operations is negligible and are therefore ignored.

To see what this would imply in practice consider the AES circuit described in [PSSW09]; which

[1] 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.

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.[2] Assuming the parties share a XOR sharing of the AES key, (which adds an additional $2 \cdot n \cdot 128$ gates and wires to the circuit), the parameters for the **Initialize** call to the $\mathcal{F}_{\mathsf{SPDZ}}$ functionality in the preprocessing-I protocol will be

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

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

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

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.

[2]Note that unlike [PSSW09] and other Yao based techniques we cannot process XOR gates for free. On the other hand we are not restricted to only two parties.

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

- In [DKL+12] 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 [KSS13] 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 [NNOB12] 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 [DKL+12], 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.

Chapter 4

Security Proof

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

To be able to follow the proof smoothly we first present some conventions and notations. In both the original BMR protocol and our protocol the players obtain a garbled circuit and a matched set of garbled inputs, they are then able to evaluate the circuit without further interaction. The players evaluate the circuit from the bottom up until they reach the circuit-output wires. I.e. the input wires are said to be at the "bottom" of the circuit, whilst the output wires are at the "top". In their evaluation the players use the garbled gate 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 gate's entry for which the players evaluate that gate as the

active entry and the other three entries as *inactive* entries. Similarly we use the term *active signal* to denote the value Λ_c that is revealed for some wire c, and the term *active path* for the set of active signals that have been revealed to the players during the evaluation of the circuit. Recall that in the online phase of our protocol the players exchange the active signal of all the circuit-input wires. We denote by I the set of indices of the players that are under the control of the adversary $\mathcal{A}$, and by x_I denoted 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 denoted 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. We denote the set of gates in the circuit as $G = \{g_1, \ldots, g_{|G|}\}$. Recall that κ is the security parameter.

4.1 Security in the semi-honest model

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

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

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

In addition the adversary sees:

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

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

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

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

In this section we present the procedure $\mathcal{P}$ and show that $\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{\bar{x}}$ and $\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})\}_{\bar{x}}$ 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.

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

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.*

[1] 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 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 $\{k^i_{w,0}, k^i_{w,1} \in \mathbb{F}_p \mid w \in W\}$.
3. **Keys extensions.** For every key $k^i_{w,b}$, and for every gate g which wire w enters into, the values

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

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.

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 the garbled entries (except one as described immediately) to be random values from $(\mathbb{F}_p)^n$ whilst for the $(2 \cdot \Lambda_a + \Lambda_b)$-th entry the algorithm instead conceals the Λ_c key-vector (in contrast to the real construction in which the key-vector that the entry conceals depends on the masking values of a, b and c). That is, when the algorithm construct the garbled gates it ignores the masking values that it chose in the previous step. For example, take $\Lambda_a = 1$,$\Lambda_b = 0$ and $\Lambda_c = 1$ then the entry by which the players evaluate the gate is the $2 \cdot \Lambda_a + \Lambda_b = 2$ (i.e. the third) entry which is C_g. Thus $\mathcal{P}$ makes C_g to encrypt the 1-key vector, i.e. $\mathbf{k}_{c,1}$ by:

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

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

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

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

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

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

then, if the following computational indistinguishability holds (proven in claim 2)

$$\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})\}_{\bar{x}} \stackrel{c}{\equiv} \{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{\bar{x}} \tag{4.1}$$

then by transitivity of indistinguishability, it follows that

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

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

In the following we prove Equation 4.1:

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

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

Proof. Remember that in the 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 couldn't compute the indicators x_A, x_B, x_C, x_D (as described in section 3.2) and thus couldn't tell which key vector is encrypted in each entry, that is, we couldn't fill out correctly the four garbled gate's entries A, B, C, D. On the other hand, in the procedure $\mathcal{P}$ we do know the set of external values

$\{exvw \mid w \in W\}$, thus, we know for sure that for every gate g, with input wires a, b and output wire c, the key vector encrypted in the $2\Lambda_a + \Lambda_b$-th entry of the garbled table of gate g is the Λ_c -th key vector $\mathbf{k}_{c,\Lambda_c}$.

Let us denote by $\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j}$ 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 $\{k^i_{w,\beta} \mid 1 \leq i \leq n, w \in W, \beta \in \{0,1\}\}$ and the masking values $\{\lambda_j \mid j \in W\}$. Similarly, denote by $\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j}$ the view of the adversary in the output of the procedure $\mathcal{P}$.

Given that

$$\{\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j} \stackrel{c}{\equiv} \{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j} \tag{4.2}$$

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

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

since the functionality, keys and masking values are taken from exactly the same distributions in both cases. In the following (claim 3) we prove that Equation 4.2 holds. □

Claim 3. *Fix a functionality f, 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 the procedure $\mathcal{P}$, then equation (4.2) 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 $\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j}$ are 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 section 3.2 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

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

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 ordering 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 the 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 (Section 3.2). 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 $\{\mathcal{P}(\text{REAL}_{\mathcal{A}}^{\mathsf{BMR}})\}_{f,\bar{x},k^i_{w,\beta},\lambda_j}$. Thus, by 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 the procedure $\mathcal{P}$, that is, the external values $\Lambda_a, \Lambda_b, \Lambda_c$ are known and thus the key $\mathbf{k}_{c,\Lambda_c}$ is encrypted using keys $\mathbf{k}_{a,\Lambda_a}$ and $\mathbf{k}_{b,\Lambda_b}$ in the $2\Lambda_a + \Lambda_b$-th entry (the active entry) while the other three (inactive) entries are independent of $\mathbf{k}_{a,\Lambda_a}$, $\mathbf{k}_{b,\Lambda_b}$, $\mathbf{k}_{a,\bar{\Lambda_a}}$ and $\mathbf{k}_{b,\bar{\Lambda_b}}$

(because $\mathcal{P}$ chooses them at random from $(\mathbb{F}_p)^n$).

- If the view is taken from H^z then the garbled table of g is computed correctly for all the four entries. Let $\tilde{g}_a$ be a gate whose output wire is a (which, 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 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 $(F_p)^n$, therefore reveal no information about $\mathbf{k}_{a,\Lambda_a}$, and more important, no information about $\mathbf{k}_{a,\bar{\Lambda_a}}$. The same observation holds for the gate $\tilde{g}_b$ whose output wire is b. We get that in the computation of the garbled table of gate g (recall that it is in layer $z+1$ and we are currently looking at hybrid H^z) there is exactly one entry (i.e. the active entry) which depends on both $\mathbf{k}_{a,\Lambda_a}$ and $\mathbf{k}_{b,\Lambda_b}$ while the other three (inactive) entries are depend on at least one of $\mathbf{k}_{a,\bar{\Lambda_a}}$ and $\mathbf{k}_{b,\bar{\Lambda_b}}$, 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 know $\tilde{k}$ at all, e.g. when working with an oracle.

In the following we exploit the above observation: since the distinguisher $\mathcal{D}$ has no information about $\mathbf{k}_{a,\bar{\Lambda_a}}$ or $\mathbf{k}_{b,\bar{\Lambda_b}}$, we could 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 could even use an oracle to 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 pseudo random function under multiple keys:

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

distinguishers $\mathcal{D}$, there is a negligible function neg such that:

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

where $F_{\bar{k}} = F_{k_1}, \ldots, F_{k_{m(n)}}$ are the pseudo random function F keyed with polynomial number of randomly chosen keys $k_1, \ldots, k_{m(n)}$ and $\bar{f} = f_1, \ldots, f_{m(n)}$ are $m(n)$ random functions 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 pseudo random function then it is a pseudo random function under multiple keys, thus, since the function F used in our protocol is a PRF then for every polynomial time distinguisher $\tilde{\mathcal{D}}$, every positive polynomial p and for all sufficiently large κ:

$$|Pr[\tilde{\mathcal{D}}^{F_{\bar{k}}(\cdot)}(1^\kappa) = 1] - Pr[\tilde{\mathcal{D}}^{\bar{f}(\cdot)}(1^\kappa) = 1]| \leq \frac{1}{p(\kappa)} \tag{4.3}$$

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 $\mathcal{D}$ who distinguishes between H^z and H^{z+1} with non negligible probability ε', we construct a polynomial time distinguisher $\mathcal{D}'$ who distinguishes between F under multiple keys and a set of truly random functions (and thus contradicting the pseudorandomness of F). The distinguisher $\mathcal{D}'$ has an access to $\overline{\mathcal{O}} = \mathcal{O}_1, \ldots, \mathcal{O}_{2n}$ (which is either a PRF under multiple keys or a set of truly random functions), $\mathcal{D}'$ act as follows:

1. Chooses keys and masking values for all players and wires, i.e. $\{k^i_{w,b} \mid w \in W, b \in \{0,1\}, i \in \{1, \ldots, n\}\}$ and $\{\lambda_w \mid w \in W\}$.

2. Constructs the gates $g_1, \ldots, g_z$ as described in the procedure $\mathcal{P}$, i.e. only the [active entry] is calculated correctly, the rest three entries are taken to be random from $(\mathbb{F}_p)^n$.

3. Construct the garbled table of gate g_{z+1} in the following manner: denote its input wires 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 computes it correctly as in our 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 our protocol.

5. Hands the resulting view to $\mathcal{D}$ and outputs whatever it outputs.

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

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

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

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

□

4.2 Security in the malicious model

When our protocol relies on SPDZ as its underlying MPC then 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 a waste 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 first kind of behavior has the same effect as if the adversary inputs to the functionality the value $\bar{\rho_w}$ instead of ρ_w, since $\bar{\Lambda_w} = \lambda_w \oplus \bar{\rho_w}$, and thus, this behavior is permitted to a malicious adversary (i.e. a malicious adversary is able to change the input to the functionality without being considered as a cheat since this behaviour is unavoidable even in the ideal model).

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

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

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 j-th element of A_g is

calculated (as described in Functionality 3) by

$$A_g^j = \left(\sum_{i=1}^{n} F_{k_{a,0}^i}(0 \| j \| g) + F_{k_{b,0}^i}(0 \| j \| g) \right) + k_{c,v}^j \tag{4.4}$$

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

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

i.e. X^j is the sum of the PRF results that the adversary provides and Y^j is the sum of the PRF results that the honest player provides. Thus, rewriting equation (4.4) we obtain

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

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

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

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

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

as required and the j-th element (which actually verified by P_j) will be flipped. Observe that in order to succeed the adversary has to find Δ^j. But, since $k_{c,v}^j$ and $k_{c,\bar{v}}^j$ are random 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 flipped-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:

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

Now, assume that when the adversary guesses a wrong Δ^j for some entry of some gate, the parties do not abort and somehow can keep evaluating the circuit using the correct key; then the probability of the adversary 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. □

4.2.2 Emulation in the ideal model.

In the following we describe the ideal model in which the adversary's view will be emulated, then we show the existence of a simulator $\mathcal{S}'_{\text{OUR}}$ in the malicious model 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 [Cle86]).

The ideal execution of f on inputs $\bar{x}$ and corrupted parties P_I (who 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 the 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 to it, e.g. a simulated view) and a set of keys $\mathbf{K_I}$, and outputs the exact same view, but rather, the keys that are opened to the adversary now are $\mathbf{K_I}$. The procedure $\mathcal{H}$ is given a view of the adversary in the semi-honest model (or a view that is indistinguishable to it) and a set of PRF results $\mathbf{F_I}$, and outputs the exact same view, but rather, 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 to their output in the real execution of our protocol.

The idea of the simulation method is that we can use the fact that there exist a simulator $\mathcal{S}_{\text{OUR}}$ 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 ($\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})$), 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 $\mathcal{P}$ 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 $\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})$ 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 the 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* $\text{REAL}_{\mathcal{A}}^{\mathsf{Our}}(\bar{x})(\bar{x})$ *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 $\bar{x}$

$$\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x}) \stackrel{c}{\equiv} \mathcal{P}'(\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x}), \mathbf{K_I}) \tag{4.5}$$

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 the 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*

$$\{\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})\}_{\bar{x}} \stackrel{c}{\equiv} \{\mathcal{P}'(\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x}), \mathbf{K_I})\}_{\bar{x}} \tag{4.6}$$

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 undelying 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:

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 3 we can see that

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

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

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

Finally, it adds that difference to the appropriate coordinate in one of the vectors A_g, B_g, C_g, D_g as described above. For instance. let $F^{\Delta}_{k^i_{a,0}}(0 \parallel j \parallel g) = \tilde{F}_{k^i_{a,0}}(0 \parallel j \parallel g) - F_{k^i_{a,0}}(0\|j\|g)$ 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'.

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*

$$\text{REAL-MAL}^{\mathsf{Our}}_{\mathcal{A},I}(\bar{x})_{\mathbf{K_I},\mathbf{F_I}} \equiv \mathcal{H}(\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})_{\mathbf{K_I}},\mathbf{F_I}) = \mathcal{H}(\text{REAL}^{\mathsf{Our}}_{\mathcal{A}}(\bar{x})_{\mathbf{K_I}},\mathcal{A}(r,\mathbf{K_I},\lambda_{\mathbf{I}})) \tag{4.7}$$

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

The simulator $\mathcal{S}'_{\mathbf{OUR}}$

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 halt the internal execution right after it receives the external values $\mathbf{\Lambda_I}$ to all the corrupted parties in the online phase (that is, it halts after Step 1 of the online phase of Protocol 1). 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 $\mathbf{\Lambda_I}$ the simulator $\mathcal{S}'_{\text{OUR}}$ can conclude $\mathcal{A}$'s input to the functionality x_I.

2. Now, focusing on the ideal world, the honest parties and $\mathcal{S}'_{\text{OUR}}$ (this time as the adversary) send their inputs to the trusted party. $\mathcal{S}'_{\text{OUR}}$ sends x_I (that 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 $x_J = 0^{|J|}$.[2] If they abort then $\mathcal{S}'_{\text{OUR}}$ instructs the trusted party to not send the output y_J to P_J (i.e. to output $\perp$). Otherwise, if the evaluation succeeds then $\mathcal{S}'_{\text{OUR}}$ instructs the trusted party to output the correct output y_J. [3]

8. The simulator $\mathcal{S}'_{\text{OUR}}$ outputs the view v'' as the adversary's simulated output.

Indistinguishability: Real vs. 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:

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

[2] Note that the correctness property 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 $x_J = 0^{|J|}$.

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

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*

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

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

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

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

$$|Pr[\mathcal{D}(\{V_{\text{REAL-MAL}}^{\mathcal{A}}(\bar{x}), O_{\text{REAL-MAL}}^{J}(\bar{x})\}) = 1] - Pr[\mathcal{D}(\{V_{\text{IDEAL}}^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}(\bar{x}), O_{\text{IDEAL}}^{J,\mathcal{S}'_{\text{OUR}}}(\bar{x})\}) = 1]| = \varepsilon(\kappa)$$

we describe a distinguisher $\mathcal{D}'$ that is able to distinguish between $V_{\text{REAL-MAL}}^{\mathcal{A}}(\bar{x})$ and $V_{\text{IDEAL}}^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}(\bar{x})$ 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 $\mathcal{D}'$ act 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 $V_{\text{REAL-MAL}}^{\mathcal{A}}(\bar{x})$ or $V_{\text{IDEAL}}^{\mathcal{A},\mathcal{S}'_{\text{OUR}}}(\bar{x})$.
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, $\mathcal{D}'$ evaluate the circuit using $\bar{x}$ and assign 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:

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

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

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

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 Equation 4.7, the second follows from 4.5 and the third follows from the operation of the simulator of the semi-honest model. That is, if there exist a distinguisher who succeed 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.

□

Bibliography

[BB89] Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In Piotr Rudnicki, editor, *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, Edmonton, Alberta, Canada, August 14-16, 1989*, pages 201–209. ACM, 1989.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10. ACM, 1988.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In Harriet Ortiz, editor, *22nd STOC*, pages 503–513. ACM, 1990.

[BNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS*, pages 257–266. ACM, 2008.

[Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 364–369. ACM, 1986.

[DKL+12] Ivan Damgård, Marcel Keller, Enrique Larraia, Christian Miles, and Nigel P. Smart. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In Ivan Visconti and Roberto De Prisco, editors, *SCN 2012*, volume 7485 of *LNCS*, pages 241–263. Springer, 2012.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS*, volume 8134 of *LNCS*, pages 1–18. Springer, 2013.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [SC12], pages 643–662.

[GHKL08] S. Dov Gordon, Carmit Hazay, Jonathan Katz, and Yehuda Lindell. Complete fairness in secure two-party computation. In Cynthia Dwork, editor, *Proceedings of the 40th*

Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008, pages 413–422. ACM, 2008.

[GMW86] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to prove all np-statements in zero-knowledge, and a methodology of cryptographic protocol design. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 1986.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *19th STOC*, pages 218–229. ACM, 1987.

[KSS13] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 549–560. ACM, 2013.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings*, volume 4515 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2007.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [SC12], pages 681–700.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.

[SC12] Reihaneh Safavi-Naini and Ran Canetti, editors. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164. IEEE Computer Society, 1982.

Appendix A

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

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

1. **Initialize the MPC Engine:** Call **Initialize** on the functionality $\mathcal{F}_{\mathsf{MPC}}$ with input p, a prime with $2^\kappa < p < 2^{\kappa+1}$.

2. **Generate wire masks:** For every circuit wire w we need to generate a sharing of the (secret) masking-values λ_w. Thus for *all* wires w the players execute the following commands

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

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

   - 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 $[\mathbf{k}_{w,j}]$.