Research Proposal

Avishay Yanay
Under supervision of Prof. Yehuda Lindell and Prof. Benny Pinkas
Bar-Ilan University

March 2015

Contents

1 Introduction

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 to be 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 the 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 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 show 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 classic works, including Yao's protocol for two-party, the GMW approach for both two-party and multi-party cases and the BMR idea for constant round protocol in the multi-party case, along with an overview of the current state of the art solution to the multiparty and dishonest majority case ([DPSZ12]). Later we present the research question and the possible directions toward the solution.

2 Background

2.1 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.

2.2 GMW methodology ([GMW87]

In contrast to Yao's solution (that 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 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 enters 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 $c_i = m a_i b_i + \sum_{j \ne i} c_i^{i,j}$ (it is obvious that $m a_i b_i = 0$ if m is even and $m a_i b_i = a_i b_i$ 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 output a result protocol that is secure under a 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 it 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 prty 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* propert 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 result 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 obtain 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 uses *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.

2.3 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 $f_g : \{0,1\}\times\{0,1\} \rightarrow \{0,1\}$), 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. Theore, 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 on 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 $L = n \cdot \kappa$ bits of the output of G by G^1, and the last $n\kappa$ bits of the output of G by G^2. 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):

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

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

Step 1 – send values:

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

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

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

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

2.4 SPDZ protocol

Damgård, Pastro, Smart and Zakarias (written [DPSZ12] and pronounced *speeds*) recently presented a practical solution to the multi-party secure computation in the dishonest majority case (with malicious adversary). Their solution however is based on the GMW paradigm, 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 are indeed requires communication rounds but unlike the GMW solution which achieves that by invoking a two-round protocol many times, the SPDZ solution achieves 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, it 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 player 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 is working in the preprocessing model, i.e. the expensive part of the computation is executed in the preprocess 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 is 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 a 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 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, beginning from the circuit-input wires, evaluating the gates, one after the other, till 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 possibility to easily add 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 later 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 is 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 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 $D_{sk_1,\ldots,sk_n}(c)$. The parties 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:
  1. Party P_i generates a random f_i and transmits $c_{f_i} = Epk(f_i)$
  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 $(j \neq i)$ 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:
  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 achieve $[b]$.

- *Generating* $[c]$. Recall that it 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 FHE scheme which makes the computation expensive while in the online phase the parties executes only simple operations, moreover, given a trusted dealer which implements the offline phase then the result protocol is information theoretically secure against up to $n - 1$ maliciously corrupted parties.

3 Research Question

The main goal of our research is to improve the results achieved by the current state of the art solutions ([DPSZ12, NNOB12]) in terms of communication complexity or underlying assumptions. As we desire to achieve a constant round protocol we would begin the research using the [BMR90] as our basis. Mainly, we want to preserve the efficiency of the BMR protocol in the semi honest model and use it to derive an efficient protocol in the malicious model. We currently consider two methodologies in which we can proceed:

1. Use [DPSZ12] as the underlying protocol in BMR (i.e. replace the usage of GMW with SPDZ); this would require to change the circuit, by which the parties compute the garbled gates, from a boolean to an arithmetic circuit. As described above, since the SPDZ protocol is resistant against any number of maliciously corrupted parties (less than n) and has a very efficient online phase, this approach might answer our requirements. SPDZ, however, has expensive computation in the offline phase, but still, it is better than using GMW (or [BGW88] as used in [BNP08]). It might be possible to use SPDZ to garble every gate independently but simultaniouely, and since the circuit that is required to do so is relatively flat, this would yield a very efficient protocol and introduce a new observation which was not shown before.
2. Another approach would be to tailor a specific protocol to achieve the secure gate garbling, although this would consume much more time and effort, this might yield a way more efficient protocol than a generic one.

One critical issue that must be addressed in both the aforementioned approaches is that when using a protocol computing a garbling independently for every gate it must be verified that the parties enters to the protocol with correct PRG (extensions) values (see description of the BMR protocol below) and that those values are consistent in every invocation of gate garbling protocol. This requirement could be captured in the following ways:

- Ensure that the parties enter the correct PRG results; this might be a hard (expensive) task, however, it allows the parties to be sure that the garbled circuit is correct in the beginning of the execution (and if a malicious behaviour detected, they would not proceed to the online phase and obviously nothing has been revealed). This approach is taken in the original BMR protocol (using ZK proofs).
- Not verifying the correctness of the computation of the PRGs; this might yield two types of security:
  1. *Malicious adversary model.* If a corrupted party is caught cheating but still nothing is revealed then the protocol is said to be secure in the malicious model.
  2. *Covert adversary model.* Otherwise, if the corrupted party succeeds in learning some information with regard to the other parties' inputs, this protocol is said to be secure in the covert adversary model.

References

[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.

[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.

[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.

[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.