Secure Multiparty Computation

- Introduction
- Settings and Definitions
- Protocols
- Code
- Conclusion
- Bibliography

Submitted by: Dor Messica, Rotem Badash, Ido Roditty.

Introduction

Multi-party computation (or MPC) is a subfield in cryptography that focuses on different computational methods that aims to enable different parties to interact and compute a joint function. It is a cryptographic technique allowing multiple parties that have private information, to perform a joint computation that combines their data without disclosing their inputs' content or any other private information in the process.

We consider a scenario where a number of distinct but connected parties want to carry out a joint computation of a function. The goal of the computation is to have each party receive the result of the function over all parties' data, without revealing any of the personal inputs to each other. The aim of secure multiparty computation is to enable the parties to carry out the required computation in a secure manner, and it concerns the possibility of malicious behavior by some adversarial entity. It is assumed that a protocol's execution may be attacked by an external entity or by a subset of corrupted participating parties, trying to learn private information or compromise the result of the computation.

Two important requirements for any secure MPC protocol are privacy - nothing should be learned beyond what is absolutely necessary (parties learn their output and nothing else) - and correctness - each party receives the correct output. Meaning, the security in this case is generally understood as guaranteeing, in the presence of malicious behavior from some parts of the system, a set of correctness properties for the output values, together with a set of security requirements regarding the private data of the participating parties.

Eventually, the goal of MPC is to work as if there is a third party member, which is responsible for getting the input from all parties and return the true value of the function, where MPC actually exists for those reasons that we can not trust and third party member for this purpose, as it can also be corrupted. The power of MPC is that If we assume secure point to point channels (and broadcast) and honest majority, we can get unconditional security for any function, and be quite effective too.

In fact, MPC unites separate entities holding pieces of information that, when combined, can reveal a secret, sign a message or approve a transaction. In this case, the participating entities do not trust each other, but wish to jointly compute a function over their inputs while keeping their data hidden from the other entities and outside parties.

Secure multiparty computation can be used to solve a wide variety of problems. Today, MPC has many applications, such as electronic voting, digital auctions, private DNA comparisons and more. One of the top applications for MPC is for securing digital assets and it has become the standard for institutions looking to secure their assets while retaining fast and easy access to them.

An example of the use of secure multiparty computation is the problem of comparing a person's DNA against a database of cancer patients' DNA, in order to find people with a high risk for a certain type of cancer. DNA information is very sensitive and should not be revealed to organizations. This can be resolved by running a secure multiparty computation that reveals only the type of cancer the examined DNA is close to. In this example, the privacy requirement ensures that no DNA information is revealed, and the correctness requirement ensures that no malicious party can change the result.

MPC's began in the 80s, when Andrew Yao, a computer scientist first adopted two-party computation and introduced secure multi-party computation.

A few milestones in the history of MPC:

- 1982 – secure two-party computation is introduced as a method of solving the Millionaire's Problem.
- 1986 – Andrew Yao adapts two-party computation to any feasible computation.
- 1987 – a group of computer scientists adapted two-party computation to multi-party computation.
- 1990s – the study of MPC leads to breakthroughs in areas including universal composability.
- 2008 – The first large scale, practical application of MPC, demonstrated in an auction.
- 2010s – MPC is first used by digital assets custodians and wallets for digital asset security.
- 2019 – MPC-CMP, an automatic key-refreshing MPC algorithm is first introduced.

Settings and Definitions

We will look at a model of multiparty computation with the following setting -

- Input parties $P_1 .. P_n$, each containing a private input x_i.
- The goal is to jointly compute a function f with the inputs - $y = f(x_1,.., x_n)$. y will become public but the computation of this value must preserve security properties such that the only information revealed on the inputs is the result y. The security properties should hold even if some of the input pirates maliciously attack the protocol.
- We will want the executed protocol to be equivalent to having a trusted party T that receives each private input from each party, computes the function f and returns the result y.

Security:

In this setting, we have an adversary controlling a subset of input parties (they will be called "corrupted parties"). Secure protocols will need to withstand any adversarial attack. In order to decide whether a protocol is secure we will need a definition for security in MPC - different definitions aiming to ensure important security properties general enough to capture most of the multiparty computational tasks, have been proposed. A few important requirements that any secure computation protocol must fulfill are:

1. Privacy - no party should learn anything more than its output. In particular, the only information that should be learned about other parties' inputs is what can be derived from the output itself.
2. Correctness - each party is guaranteed to receive the correct output.
3. Independence of inputs - each party must choose its input independently of the other parties' inputs (including the corrupted parties).
4. Guaranteed output delivery - parties shouldn't be able to prevent other parties from receiving their output. In other words, the adversary should not be able to disrupt the computation by carrying out a "denial of service" attack.
5. Fairness - parties should receive their outputs if and only if the other parties also receive their outputs. The scenario where a corrupted party obtains output and another party doesn't shouldn't be allowed to occur.

Note - the list above does not constitute a security definition but rather a set of requirements that should hold for any secure protocol.

There are different ways to define security. One option is a **property-based** definition - defining a list of requirements for the task (such as the list above) and saying that a protocol is secure if it fulfills all the requirements. This approach isn't sufficient - it is hard to know if we missed important necessary requirements, especially because different applications need different requirements and we would like a general definition to capture as many applications as possible. Furthermore, a definition should be simple enough so it's easy to see that all the possible adversarial attacks are prevented by it. In addition, it is difficult to analyze complex tasks and reduce their complexity to a list of requirements.

Another option is **the real / ideal paradigm** - everything an adversary can achieve by attacking a real protocol, can be achieved by attacking an ideal computation involving a trusted party. We consider a "real world" where we have a trusted third party for the required computation - the input parties send their inputs to the

trusted party, who computes the function and returns each party its output. In this scenario the only freedom an adversary will have is to choose the input a corrupted party will send. In the real world we have no such trusted party - the input parties run a protocol amongst themselves. A secure protocol will be a protocol that can emulate the described "ideal world". It will be considered secure if no adversary can harm the real execution more than it can harm the ideal world execution. For any adversary successfully carrying out an attack in the real world, there exists an adversary successfully carrying out an attack in the ideal world with the same effect.However, in the ideal world an adversary cannot carry a successful attack so we conclude that there cannot be an adversary carrying a successful attack in the real world.

In this approach we are using a simulated-based security. In this case we will simulate the ideal world using a simulator $\mathcal{S}$, while in the real world we will have an adversary $\mathcal{A}$. We will use a distinguisher $\mathcal{D}$ that will perform the following steps: first, $\mathcal{D}$ will give inputs to the parties. Then, it will get back outputs from the parties and from an adversary / simulator. In the end $\mathcal{D}$ will guess if we are in the real world (used an adversary) or in the ideal world (used a simulator). Meaning $\mathcal{D}$ will try to distinguish an adversary from a simulator. A protocol will be considered secure if for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$, such that every distinguisher $\mathcal{D}$ can successfully distinguish between $\mathcal{A}$ and $\mathcal{S}$ with a negligible probability.

The advantages of this approach are that it is very general and can capture any computational task. It is simple to understand and very useful for modular design protocols. Furthermore, this approach does not miss any needed security requirements.

The real / ideal paradigm supports **sequential and concurrent modular composition** - in reality, a secure multiparty protocol runs as a part of a system. It was proven that the MPC protocol will still behave as if a trusted party runs the computation (modular composition theorem). This enables constructing large protocols using secure subprotocols and analyzing a large system that uses MPC protocols. In the sequential composition setting, the MPC protocol can run as a subprotocol of another, but it must be run without other messages being sent in parallel - stand-alone. The stand-alone setting that is being considered by the basic security definition, is when there is a single protocol execution at any given time. In this case the theorem states that the MPC protocol behaves as if a trusted third party carries out the computation, but in many cases the MPC protocols run at the same time as other instances of themselves or other protocols run which means that a protocol proven secure under the stand-alone setting may not be secure in this case. These cases are defined as the concurrent general composition where we have a number of arbitrary protocols that are executed concurrently. This requires a stronger definition, and the most popular security definition suggested for this setting is the universal composability (UC) - we consider a protocol P and a function F, where P's parties make calls for F with input values and P has instructions on what to do with F's outputs. We allow multiple occurrences of F, where its sessions run independently. Protocol π that UC-realizes F, and we compose a protocol $P^{F\to\pi}$ (an input to F is given to a machine in the corresponding session of π and it's outputs are given to P and treated as outputs of F). The universal composition theorem states that running this protocol is "at least as secure" as running P - guarantees that for any adversary A there exist an adversary S such that interacting with A and parties running $P^{F\to\pi}$ is indistinguishable from interacting with S and parties running P with a negligible probability. This means that if P UC-realizes an ideal functionality G, then so does $P^{F\to\pi}$. Any protocol proven secure according to this definition is guaranteed to behave like an ideal execution, irrespective of what other protocols run in parallel. However, this comes at a price of efficiency and assumptions required on the system setup.

We can consider designing the protocol in a hybrid model (similar to the stand-alone setting in the real world), where we have a trusted party that computes the functionality F when there are no other messages

sent. The sequential modular composition theorem in this case will say that if protocol π securely computes functionality G in a F-hybrid model, and protocol ρ securely computes F, then the protocol π^{ρ} securely computes g in the real world.

An important note is that the MPC secures the process, but not the outputs - nothing is revealed during the computation itself, but that does not mean that the output of the function that was computed does not reveal sensitive information. This means that while the MPC secures the computation process, which functions should be computed still needs to be considered .

There are three security types:

- Computational - we will use a PPT distinguisher, meaning we would like the real world and the ideal world to be computationally indistinguishable
- Statistical - The distinguisher will have a negligible error probability, meaning the real world and the ideal world will be close
- Perfect - The distinguisher will have aero error probability, meaning the real world and the ideal world are identically distributed.

Adversarial Model:

In addition to the described above, we will need to address the main parameters defining the adversaries attacking the protocols and their power:

- **Adversarial behavior/power**:

  A malicious attack in this scenario is modeled by an external adversary *A* that corrupts some of the input parties and controls their actions. We will consider 3 types of allowed adversarial behavior:

  - Semi-honest adversaries - the corrupted parties will follow the protocol, while the adversary learns their internal states (including messages correspondence) and tries to use it to learn the private information. This is considered a weak adversary but a protocol that is secure from this type of adversary guarantees that there isn't internal data leakage between the input parties.
  - Fail-stop adversaries - the corrupted parties will follow the protocol, but will be able to prematurely stop. Such adversaries model crash failures.
  - Malicious adversaries - the corrupted parties can deviate from the protocol, according to the adversary decisions. These adversaries are called "active" and providing security against them ensures no adversarial attack can succeed.
  - Covert adversaries - the adversary will attempt to break the protocol. The security guarantees that the adversary will be detected with a specified probability (according to the application). If it is not detected, the adversary will be able to learn private information and cheat the application.

- **Adversarial power:**
  - Polynomial time - computational security, the computation time is bound to be polynomial in the input length. This usually requires cryptographic assumptions (encryption, signatures…)
  - Computationally unbounded - information thoracic security - an all-powerful adversary with no bound on the computation time.

- **Corruption strategy:**

  there are different models of the corruption strategy used by the adversary, that describes when input parties are corrupted and how:

  - Static corruption model - the adversary controls a fixed set of parties, defined before the protocol's execution begins. In this case honest parties remain honest during the entire execution.
  - Adaptive corruption model - adaptive adversaries have the capability of corrupting the input parties during the function computations. The adversary chooses arbitrarily which input parties to corrupt and when, and it may depend on its view on the computation execution. This models a party that is originally honest and during the execution changes his behavior to dishonest (once a party is corrupted it remains corrupted).
  - Proactive security model (mobile adversary) - in this case parties can be considered corrupted for a certain period of time only, which means honest parties can become corrupted during the execution and corrupted pirates can become honest during the execution. This scenario may happen when considering an external adversary that breaks into systems, networks or devices but loses its control when the systems are cleaned. Securing from this kind of corruption model promises that an adversary will only be able to learn private information from the local state of the corrupted device and only while it is corrupted.

- **Number of corrupted parties:**
  - Threshold adversary - denoted by an upper bound on the number of corruptions. When n is the number of parties, and t is the upper bound we have:
    - No honest majority
    - Honest majority - $t < n/2$
    - Two-thirds majority - $t < n/3$
  - General adversary structure - the protection will be against specific subsets of parties.

Network / Communication model:

We will also need to consider the Network / Communication model that will be used:

- **Communication model:**
  - Point-to-point - a fully connected network of channels between pairs of elements.
  - Broadcast - we will have a broadcast channel such that every message sent on it reaches all the connected elements.

- **Message delivery:**
  - Synchronous - the protocol proceeds in rounds, such that every message arrives within a known time frame.
  - Asynchronous - the adversary can impose a (finite) delay on any message

- Fully asynchronous - the adversary has full control over the network and will be able to drop messages.

Settings and the Ideal world:

Let's recall the ideal world we described:

1. Each party sends its input to the trusted party
2. The trusted party computes $y = f(x_1,.., x_n)$
3. The trusted party sends y to each party

This is an overly ideal world sense we cannot always achieve fairness (only with an honest majority). So we can suggest an ideal world without fairness:

1. Each party sends its input to the trusted party
2. The trusted party computes $y = f(x_1,.., x_n)$
3. The trusted party sends y to the adversary
4. The adversary responds with continue / abort
5. If he responded with continue then the trusted party sends y to all the parties, if it responded abort the trusted party sends $\perp$ to all the parties

In this world we satisfy correctness, privacy and independence of inputs.

There are two main settings:

- **Computational setting** - in this setting we will have an arbitrary number of corrupted parties (unlimited), the communication channels will be authenticated and we will have computational security using PPT adversary and a distinguisher.
- **Information-theoretic setting** - in this setting we will have an honest majority (number of corrupted parties will be bound by $n/2$ or $n/3$), the communication channels will be secure and we will have statistical / perfect security using an all powerful adversary and a distinguisher.

Feasibility of MPC:

It has been established that any distributed computing function can be securely computed when there are malicious adversaries. Let n denote the number of parties, and let t denote a bound on the number of corrupted parties -

- For $t < n/3$, every function f can be computed with perfect security, meaning secure multiparty protocols with fairness and guaranteed outputs can be achieved for any function f with computational security assuming synchronous point-to-point network and authenticated channels, or with information-theoretic security assuming private channels.
- For $t < n/2$, every function f can be computed with statistical security, meaning secure multiparty protocols with fairness and guaranteed outputs can be achieved for any function f with computational and information-theoretic security, assuming the parties have access to a broadcast channel.
    - For Semi-honest setting every function f can be securely computed with perfect security.

- For $t \geq n/2$ (no limit on the number of corrupted parties, same as saying - $t < n$), every function f can be securely computed with abort and computational security, meaning secure multiparty protocols without fairness and guaranteed outputs can be achieved for any function f, assuming oblivious transfer.
    - For Semi-honest setting every function f can be securely computed with computational security.

OT = oblivious transfer - in the basic form, Alice (sender) can send 2 bits to Bob (receiver) via the channel, and Bob will choose one of them to receive (Alice will not find out which bit Bob chose). In general, the sender transfers many pieces of information to a receiver, and doesn't know which of them have been received. OT is a central protocol in many constructions for secure multiparty computation.

Protocols

Yao's 2PC Protocol and BMR Protocol:

In 1982, Andrew Yao introduced the importance of the MPC protocols by presenting the Millionaires' Problem: Suppose there are two millionaires, Alice and Bob, who want to find out which of them is richer without revealing their actual wealth to each other. They don't want to trust a third party to perform the comparison for them. How can they determine who is richer while keeping their individual wealth a secret? This problem is a classic example of a problem that can be solved using secure multi-party computation (MPC) protocols. In this case, an MPC protocol can be designed such that Alice and Bob each input their wealth to the protocol, which then outputs the result of the comparison without revealing the actual wealth of either party. Several different MPC protocols have been proposed to solve Yao's Millionaires' Problem, each with different trade-offs in terms of security, efficiency, and complexity.

Yao's Protocol was the first protocol for secure computation, which initially aimed for the two party case that wishes to compute some function securely. One important property of this protocol, which was also adopted later by the BMR protocol, is that it runs for a constant number of communication rounds, regardless of what the function is that we want to compute. Moreover, it is proven to be secure against semi-honest adversaries. This protocol is used for general secured computation, meaning it can compute any function at all, mainly by taking the function and representing it as a boolean circuit. These days we can efficiently compute circuits containing billions of gates, so if we want to compute a certain function effectively, we first need to check that we can represent it as a circuit of that size, and if so we can efficiently compute it securely.

This protocol works basically by setting values to the input gates, and for each gate compute the output with the appropriate truth table. But as for the secureness, we want to hide all the intermediate values and only reveal the final output of the circuit. For this reason, this protocol will act as a compiler, which takes a circuit and transforms it to a circuit which hides all information but the final output.

The main tool we will use in Yao's protocol is Garbled Circuit. This is a circuit where each gate is encrypted, and for each wire of a gate we will have two keys ("garbled values") associated with 0 and 1, and it will have the following property: For any gate, given one key on every input wire, it is possible to compute the key of the corresponding gate output, and nothing else. The tool we will use here is Oblivious Transfer, that allows a sender to send one out of several secret messages to a receiver, without the sender knowing which message was selected, and the receiver receiving only the selected message. The OT protocol ensures that the sender remains oblivious to the receiver's selection and the receiver remains oblivious to the unselected messages. Formally, in our case we have the sender (the circuit garbler) who has two of secret key $K = \{k_0, k_1\}$, the receiver (the circuit evaluator) has value b (bit) equals to 0 or 1, and the receiver obtains only the fitted key k_b with no other information being learned.

For the construction of the circuit, given a boolean circuit we assign independent random keys ("garbled values") to each wire of a gate (for example key k_0 for 0, key k_1 for 1), and encrypt each gate so that given one key for each input wire, it is possible to compute the appropriate key on the output wire using a truth table containing the key values. Moreover, in a garbled gate we change the content of the table so instead of a table

containing just the garbled values, it contains an encryption for that garbled output value with the two keys corresponding to the input values. It is important to note that in order to hide for the computing party which internal values the specific entry corresponds to, we sent the table of the gate in a permuted order, so that the party evaluating the circuit has no idea if it obtained the 0 or 1 key. At the end of the computation, we also provide a translation table for the output wire of the circuit, for the value we should reveal for the party evaluating the circuit. Overall, in this way nothing but the final output is learned for all participating parties. One more important issue that needs to be clarified is how the second party is able to evaluate the proper key for each wire while remaining the information in the circuit hidden. The solution for that matter is that for each wire of Party 2 input, the parties run an oblivious transfer (OT) protocol, where it eventually learns from Party 1 the proper key, without it knowing the key that was chosen. The OTs for all input wires can be run in parallel, reducing the overall computation time required for the protocol.

With that being said, the Yao's protocol works as follow: For inputs x and y with a fixed length n, Party 1 generates a garbled circuit G_1 with specific keys on each wire, Party 1 sends to Party 2 G_1 and the keys for all wires, both parties run all n OTs in parallel, and finally given the keys Party 2 computes G_1 and obtains the results and sends it to Party 1.

One question that comes up is how does the party computing the circuit know that it decrypted the correct entry. We know that a gate table has four entries in permuted order, and the keys known to the evaluator can decrypt only a single entry, but symmetric encryption may decrypt correctly even with incorrect keys. We introduce here two possibilities to solve this problem:

- One possibility to add redundant zeros (let's say in a size of 64 bits) to the plaintext before the encryption, so that only correct keys give redundant block, when the possibility that another entry will end with this block of zeros is 2^{-64}. The first drawback of this solution is that we have to encrypt not just the garbled values but with the concatenation of the long string of zeros, which makes the encryption much longer and will cost in communication, and makes it now the main bottleneck of the secure computation. The second drawback is that we might have to try and decrypt all four entries of the gate until we get to the right value, which can increase our computation.
- The second possibility we have is the following: We will associate each wire with a random signal bit together with the key which will remain a secret, and there will also be an "external value" bit for the wire which will be public so that the evaluator is able to see it. The property of those values is that if the signal bit is equal to 0 than the external value is equal to the internal value of the wire, and if the signal bit is 1 then the external value will be the opposite of the internal value, or in other words the external value is equal to the xor of the internal value and the signal bit. Finally, the table is going to be ordered based on the external values, so now the party evaluating the circuit that already knows the external values of the input wire will just go to the entry corresponding to those in the table and will decrypt it. It's important to mention that what we encrypt inside is not only the garbled value of the output wire but also the external value of that output wire, and that way will know the external values for the output wire for the gates in the next levels of the circuit. This way the evaluator knows exactly which entry to decrypt, and the garbled values being decrypted are being concatenated with just a single bit, so we save a lot of communication as well.

For the security of the protocol, we'll divide our reference into two parts:

- If Party 1 is corrupted (semi honest), because its view consists only of the messages it receives in the oblivious transfers, essentially if we assume the OT to be secure if we work for example when the OT

is implemented by a trusted party, than the adversary doesn't receive any information about messages, but rather just participate in those transfers.

- If Party 2 is corrupted, we can show that by only using the input and final output of the circuit, the evaluator is able to construct the circuit. Because all the evaluator learns in the protocol is the circuit itself, if we show that property we can basically conclude that everything he learns in the computation can be evaluated from the input and the output, meaning he doesn't learn any secret information. The way to prove that is to construct an additional circuit and show that Party 2 can not distinguish between the original circuit and the fake one that we construct. The fake garbled circuit property for that matter will be that the circuit always returns the final output of the original circuit. In this case, each table in the circuit will just encrypt a single output value in all 4 entries, and this ensures that no matter the input, the same known garbled values on the output wires are received.

For the efficiency of the protocol, we see that we execute 2-4 rounds of communications depends on OT and if Party 1 receives the output, the number of OTs depends on the size of the input of Party 2, and for each gate we need to use 8 encryptions in order to generate the garbled table and 2 decryptions to evaluate it. Overall everything is linear in the size of the circuit making it quite efficient .

If we suppose that OT is secure and we do have malicious adversaries, corrupted Party 1 still can not learn anything, thus we still have privacy, and it is also possible to prove full security against corrupted Party 2. The main issue is that in the case we want to make sure that the function that is evaluated is the right one, Party 1 can design a specific circuit that will make him obtain private information from Party 2 input by applying edge cases, such that when Party 2 is telling Party 1 that he is unable to decrypt some message, and for that reason it is more complicated to obtain full security against a malicious Party 1.

The Donald Beaver, Silvio Micali, Phillip Rogaway (**BMR**) protocol is a multi-party adaptation of Yao's Garbled Circuit, whereby the fundamental concept entails utilizing an MPC protocol to create an n-party garbling, followed by autonomous circuit evaluation by each player. To achieve active security, it is imperative to utilize an actively secure base MPC protocol, in conjunction with a methodology to expose any deviations from the protocol during the evaluation phase. In contemporary BMR protocols, the underlying MPC protocol typically relies on the n-party variant of Tiny-OT. The protocol was first published in 1990, and a nice property of this protocol is that it runs in a constant number of communication rounds O(1), regardless of the depth of the boolean circuit.

This protocol assumes n parties, and as in Yao we are going to associate two keys (which is being called here 'seeds') to each wire of the gate, and the important property here is that every one of the seeds is going to be a concatenation of seeds choosing by each of the n parties, in contrast to the suggestion in Yao's protocol where the one party who constructs the circuit chose the keys for all of the wires. For this reason, the parties securely compute together a table for each gate in parallel (therefore overall O(1) rounds), where given two seeds for the input wires the table reveals the seed of the resulting value of the output wire.

Essentially, the BMR protocol has two phases: The first one is a pre-processing MPC where the parties together construct the gates. Then in the second phase they provide their inputs to the actual computation of the function they want to evaluate, and then they evaluate the gates that they computed in the first phase. In other words, Given the tables and the seeds of the input values, compute the circuit as in Yao.

Let us call the the input wires a and b and the output wire c, so wire a for example has seeds $s_{a,1}^{\ 0}, s_{a,1}^{\ 1}$ chosen by Party 1, and so on each party choose two seeds for that wire $s_{a,2}^{\ 0}, s_{a,2}^{\ 1}, \ldots, s_{a,n}^{\ 0}, s_{a,n}^{\ 1}$. Each wire has a secret

bit λ, and its property is that if $\lambda_a = 0$ then $s_{a,i}^{0}$ corresponds to an internal value of 0 and $s_{a,i}^{1}$ corresponds to an internal value of 1, and otherwise the opposite. The values of λs are random and are kept hidden from the parties, by the fact that they are shared between the parties, and in that way they don't know to which internal value the 0 seeds correspond, so they actually learn nothing on the actual value going on that wire. So overall, for each gate,the table encrypting the outputs of the gate is a deterministic function of $\lambda_a, \lambda_b, \lambda_c$, the seeds as a concatenation for wires a, b and c and the gate type.

In BMR protocol, secret sharing splits a value into multiple shares, and performing arithmetic operations on these shares requires additional computations to reconstruct the final result. Specifically, when multiplying two secret-shared values, each party computes a matrix multiplication of the shares they hold to produce a set of intermediate shares that can be combined to obtain the product of the original values. The matrix multiplication involves multiplying the shares of the two values in a pairwise manner and summing the results. Similarly, when dividing a secret-shared value by a public value, each party computes a matrix inversion of the shares they hold to obtain the inverse of the public value as a set of intermediate shares. The inverse shares are then combined with the shares of the secret value to obtain the quotient of the division.

The computational complexity of the BMR protocol is proportional to the size of the circuit that needs to be evaluated. Specifically, the BMR protocol has a communication complexity of O(n^2), where n is the number of parties involved in the computation. This is because each party needs to send O(n) messages to each other party in the computation. In addition to the communication complexity, the BMR protocol also has a computational complexity of O(n^3) in the worst case, where n is the number of parties. This complexity arises from the need to perform matrix multiplications and inversions during the computation as we mentioned above. Overall, the computational complexity of the BMR protocol makes it suitable for small to moderate-sized computations, but it may become impractical for large-scale computations due to its high computational and communication costs.

The BMR protocol achieves this security guarantee mentioned above for semi-honest adversaries through the use of secret sharing and secure message transmission. The protocol involves each participant secretly sharing their input values with the other participants, and then performing a series of secure computations on these shared values to obtain the final output value. The secret sharing scheme ensures that no single participant can reconstruct the original input values of any other participant, and the secure message transmission ensures that the intermediate values of the computation are kept secret from all participants except those who are meant to learn them. However, it is important to note that the BMR protocol does not provide security against malicious adversaries who may deviate from the protocol execution in arbitrary ways to learn more information than they should from the protocol output.

To sum everything up, Those protocols can compute any functionality securely in presence of semi-honest adversaries relatively efficiently, for circuits that are not too large.

GMW:

The protocol described herein is a classic multi-party computation (MPC) protocol proposed by Goldreich, Micali, and Wigderson (GMW). This protocol employs a boolean-circuit representation for the computation and provides security against a semi-honest adversary that controls any number of corrupted parties. The basic version of the protocol, which assumes semi-honest behavior, is based heavily on Oblivious Transfer to execute the boolean gates. It is noteworthy that the GMW protocol is distinct from the GMW Compiler, which demonstrates the conversion of a semi-honest secure protocol into a maliciously secure one by utilizing zero-knowledge proofs to prevent any party from deviating from the correct protocol. The term GMW is often used to refer to a protocol that combines both techniques into an n-party MPC protocol that is actively secure.

Define Parties $P_1,...,P_n$ with inputs $x_1,...,x_n$. Now, for any function we wish to compute, we first express that function as a boolean circuit as discussed in Yao Protocol, and suppose this circuit is implemented only by XOR and AND gates (addition and multiplication modulo 2), which essentially can be used to compute any function. Moreover, the adversary might control any subset of the parties, where at the worst case it might be controlling n-1 parties, and wants to learn the input of the last party, where this subset is defined before the protocol begins ("non adaptive security"). We also assume that the communication is synchronous and there are private channels between any pair of parties, so that they can communicate without anyone being able to tap this communication, which of course can be easily implemented by appropriate tools.

For the first step of the protocol, each party shares its input bit among all other parties. So for party P_i with input bit x_i chooses random bits $r_{i,j}$ for all i≠j and sends bit $r_{i,j}$ to P_j, any then is sets its own share to be $r_{i,i} = x_i + \Sigma_{j\neq i} r_{i,j} \; mod \; 2$, so that $\Sigma r_{i,j} = x_i \; mod \; 2$. Eventually every P_j has n shares, one for each input x_i of party P_i.

The second step is to scan (evaluate) the circuit gate by gate, but the pre-defined order of the wires. P_i has shares a_i, b_i and must get share c_i of the output wire c. The simpleset case is that of addition gates (XOR), where P_i computes $c_i = a_i + b_i$, so no interaction is needed for all additional gates. The case of multiplication gates (AND) is more complicated, where P_i will obtain a share of $a_i b_i + \Sigma_{i\neq j}(a_i b_j + a_j b_i)$. Computing $a_i b_i$ by P_i is very easy since both values are known to P_i, and for the other part P_i will need to run a protocol with all other P_j: For some random bit $s_{i,j}$, P_i outputs $a_i b_j + a_j b_i + s_{i,j}$, P_j outputs $s_{i,j}$, and P_j computes four possible outcomes of the output of P_i depending on the four options for P_i's inputs (each factor a_i, b_i can be either 0 or 1). P_j eventually gets here 4 values, and then P_i and P_j run an oblivious transfer such that P_j offers P_i to learn 1 out of 4 values, which can be implemented easily by 2 basic OT. P_i is the receiver in the OT protocol, with input $2a_i + b_i$, that after each pair of parties runs this protocol (n choose 2) we will get the wanted values.

The last step is to recover and publish the outputs. The protocol is going to eventually compute shares for each output wire, and then if a certain output wire belongs to certain party who should learn the output on that wire, than all parties will send their shares of that wire to that party, and he can just sum the shares and learn its output, as we assume the parties are semi-honest.

In GMW, we want to prove security again semi-honest adversaries, and a definition for that purpose would be that given input and output, we can generate the adversary's view of a protocol execution, which basically

means that the adversary doesn't learn anything in the execution that he couldn't learn from just seeing his input and output. This can be shown by using a simulation, when we suppose in the worst case that an adversary controls the set of all parties except for one and tries to attack that party, meaning formally that the simulator is given (x_j, y_j) for all $P_j \in J$, for inputs x_j and output y_j.

Regarding the performance of the protocol, it has been determined that each multiplication gate necessitates the execution of an oblivious transfer by the parties involved. Specifically, every pair of parties (n choose 2) must utilize an oblivious transfer for each multiplication gate. The fundamental implementation of oblivious transfer involves public key operations, which renders it considerably expensive in terms of computation. This complexity surpasses that of Yao's protocol, which employs symmetric key encryption. Furthermore, in this protocol, the parties are required to establish communication between each pair for every multiplication gate to exchange messages. Although it is possible to evaluate multiple multiplication gates in parallel when they are on the same level, the evaluation of some multiplication gates must occur prior to others when they serve as inputs to subsequent multiplication gates. Therefore, the number of rounds of multiplications in this protocol is linear in the depth of the circuit, except when addition gates are involved and no interaction is necessary. This number of rounds is significantly greater than that of Yao's or BMR's protocols, where the number of rounds is typically a small number or constant, indicating a superior performance in comparison.

Oblivious transfer is considered to be very efficient, but still requires exponentiations per transfer, so when willing to make a lot of OTs, as in GMW protocol, it becomes quite inefficient. In order to make an improvement in the performance, we can improve the complexity of the OT protocol, by making it use symmetric key operations instead of public key operations with oblivious transfer extensions. In 1989, Impaglizzo and Rudich proved that there is no black box construction of OT from a one-way function, meaning we can not prevent making exponentiations, and so it isn't possible to implement it by using symmetric key encryption alone . OT extension is a protocol that uses a small number of base OTs (using public key crypto), and uses cheap symmetric key crypto to achieve many more OTs that we need.

The first construction of OT extension was due to Beaver, and it can be constructed in the following way:

For sender P_1 (and constructor of the circuit) and receiver P_2, say we want to implement n real oblivious transfers and m oblivious transfers using symmetric key crypto, where $m >> n$. So P_1 first sets his inputs to the m OTs we want to do (2 inputs for each wire), and the inputs of P_2 for that circuit is a random seed , with a length of n. We first see that in this case we indeed perform only n oblivious transfers for all inputs of P_2, which obtains the effectiveness we want. The circuit works as follows: The circuit implements a PRG (Pseudo Random Generator) that takes the seed within the input of P_2 and expands it to a size of m. Then, the circuit will have m outputs, for those that P_2 should have for the m OTs, and each output i will be the i'th bit of the output of the generator and the input of P_1 which is associated with it. It is important to notice that this implementation of circuit uses a big number of gates for each output bit, but it shows we can implement this extension using only a small number of OTs, as required. This technique makes use of Random OT, which is different from a regular OT in the way that the receiver's input bit is randomly chosen. Moreover, the security of the protocol is preserved for both corrupted receiver and sender. For the efficiency of the OT extension, as we said it's good in the sense that we only have to do n OTs, which reduces significantly the number of exponentiations, while being able to perform millions of evaluations, but this extension is much more complicated to implement.

SPDZ Protocol:

SPDZ ("Speedz") is a protocol in the dishonest majority section of secure multiparty computation (MPC), which efficiently utilizes a form of homomorphic encryption. The acronym SPDZ stands for the names of the three researchers who developed the protocol in 2012: Nigel Smart, Valerio Pastro, Ivan Damgard and Sarah Zakarias.

Given the intricacy and extensive scope of the topic of SPDZ, it is deemed appropriate to present it in a distinct (or even few) work. Nevertheless, because this protocol is scalable and provides strong security guarantees, and so has inspired numerous advancements in the field of secure MPC, including faster and more efficient protocols that build on its idea, we decided it is important to present this protocol while explaining about MPC. Therefore, we'll focus here on the main concepts of this protocol and present a general but purposeful view of its process. It is noteworthy to mention that there exist numerous variations of the SPDZ protocol in contemporary times, whereby the demonstration thereof may comprise potential arrangements that are amenable to modification into alternative schemes that ultimately generate parallel outputs."

When referring to SPDZ, it is assumed that up to n-1 out of all n parties might be corrupted at any point of the protocol, while they may also play a role of active adversary and not just passive, which means we cannot have unconditional security. This is obviously a situation that makes this protocol be focused on more edge cases and not just simple and obvious ones as other basic protocols of MPC, and therefore will be required to operate with verification steps over the correctness of the final output. It is important to mention that for this reason, the protocol suffers from two main disadvantages: The first one is that it can not guarantee termination of the protocol, because it requires the parties to be present for the computation to proceed, and if one or more parties fail to participate, drop out of the computation or even intentionaly delay or refuse to response, then the protocol cannot proceed, and the computation may be terminated. The second disadvantage is that we may be computing wrong values, as if a player is actively dishonest, and we will not know about that until the verification at the very end of the protocol's process.

The use of this protocol makes two essential assumptions: One being that there is access to synchronous communication between the participating parties, and second is that we use secure point-to-point channels, which both can be delivered by using other cryptographic tools which are not being discussed at this protocol. Moreover, the intent of the protocol is to further optimize computational complexity in MPC solutions, and to securely compute arithmetic circuits over any finite field F_{p^k} for a prime number p.

SPDZ is a kind of protocol that uses the pre-processing model of MPC, meaning it starts with a pre-processing phase, which is entirely independent of the online phase nor the inputs, and its purpose is to prepare all the necessary data in the form we need for the online phase. The advantage of employing this stage lies in its pre-execution feasibility, wherein every participating entity engages in an interactive protocol, thus rendering the subsequent online phase considerably more efficient.

The form of the SPDZ protocol is based on the use of somewhat homomorphic encryption (SHE) by each of the parties, in order to encrypt their input shares before sharing them with the other parties, which takes place in the pre-processing phase. This allows for efficient computation of certain types of functions, such as linear functions, that can be computed using simple arithmetic operations on the encrypted input shares, while still preserving their privacy. The advantage of the homomorphic encryption approach is that interaction is only needed to supply inputs and get output.

SPDZ also uses additive secret sharing, which is a specific variant of Shamir's secret sharing scheme. This is a type of secret sharing scheme in which each party splits their private input x into multiple shares x_1,..., x_n, such that the sum of the shares is equal to the original input value ($x = x_1$,..., x_n), and distributes those shares among the other parties such that each party holds a share of every other party's input. The parties then perform operations on the shares to jointly compute a function over their inputs. The shares are chosen in such a way that the private input can be reconstructed only when a sufficient number of shares are combined, and also provides strong security guarantees against malicious parties, as needed in this protocol.

Subsequent to the exposition of the aforementioned tools employed in the SPDZ protocol, the procedural steps pertaining to the preprocessing phase are delineated as follows:

1. Each party generates a public-private key pair that will be used for encryption and decryption.
2. Each party generates a set of enough random values called “pairs” and “triples”, which enable secure computation of arithmetic operations without revealing any private inputs to the other parties involved in the computation.
3. Each party splits their private input into multiple shares and distributes those shares (after being encrypted) among the other parties using additive secret sharing.
4. Each party verifies the correctness of the other parties' pre-processing steps by verifying that their shares are valid (using zero knowledge proof) to ensure that the inputs and randomness are correctly generated and shared.

Once the pre-processing phase is complete, the parties can proceed to the actual computation phase, where they jointly compute the desired function over their private inputs.

The use of triples and pairs is a crucial component of the SPDZ protocol. The encrypted shares are used to perform arithmetic operations in the online phase, such as addition and multiplication, but when directly performing these operations we might reveal information about the secret shares. To explain its use, suppose that two parties, P_1 and P_2 , wish to compute the product of two secret-shared values x and y. In the preprocessing phase, a set of triples [(a, b, c) such that $a * b = c$] is constructed randomly and shared securely between P_1 and P_2. To compute the product of x and y, P_1 and P_2 each compute (in few communicated steps) their share of the preprocessed triple, denoted by a_1, b_1, a_2, and b_2 respectively, such that $a = a_1 + a_2$ and $b = b_1 + b_2$, that correspond to shares of their secret-shared values where $x = x_1 + x_2$ and $y = y_1 + y_2$, respectively (same behavior is being used for this matter when addressing more than two parties). Both parties exchange their shares of a and b and shares of their inputs, and eventually use the precomputed triple (a, b, c) to compute the product of x and y by adding their shares of a and b to the shared value c. The final result of the computation is the secret-shared value of the product of x and y, denoted by $xy = xy_1 + xy_2$ (each computes by one party) which is known only to P_1 and P_2. The use of precomputed triples and pairs enables the efficient computation of the product of secret-shared values, without the need for costly operations during the online phase of the protocol. Specifically, the triple (a, b, c) enables the computation of the product of secret-shared values by reducing it to addition and multiplication operations on the precomputed values, while the pairs enable the computation of the sum of secret-shared values by reducing it to addition operations on the precomputed values. This approach allows for efficient computation of functions on secret-shared values, while maintaining their privacy and security.

SPDZ protocol is optimized to improve the efficiency and scalability of secure multiparty computation (MPC) protocols. Traditional MPC protocols can be computationally expensive and require a large amount of communication between parties, making them impractical for many real-world applications. The SPDZ protocol addresses these challenges by using a combination of techniques to optimize the efficiency and scalability of the protocol. These optimizations include:

- Preprocessing: the offline phase that was introduced earlier, where parties can generate precomputed values that can be used in the online phase.
- Secret sharing: The SPDZ protocol uses a secret sharing scheme to share private inputs among the parties, which allows each party to perform local computations on their shares, reducing the amount of communication required between parties.
- Homomorphic encryption: The SPDZ protocol uses homomorphic encryption to allow parties to perform computations on encrypted values without decrypting them. This technique reduces the need for expensive encryption and decryption operations, further improving the efficiency of the protocol.
- Parallelization: The SPDZ protocol allows parties to perform computations on their shares in parallel, reducing the overall computation time required for the protocol.

These optimizations allow the SPDZ protocol to achieve significantly better performance than traditional MPC protocols, making it more practical for a wide range of real-world applications. As a result, the SPDZ protocol has been widely adopted in both academic and industrial settings as a highly efficient and scalable MPC protocol.

The main idea of the online phase is to authenticate only the shared value itself and not all intermediate shares, since it is much more efficient while still providing the necessary security guarantees. Authenticating all shares individually can be computationally expensive and can require a significant amount of communication between the parties. Instead, SPDZ authenticates only the shared value, which is the sum of all shares. It can be shown that it is sufficient to ensure that the shares are authentic and that the computation is performed correctly, without requiring the parties to authenticate each share individually. In the online phase, the parties use the pre-computed values from the offline phase to jointly compute the desired function on their private inputs. The online phase involves several rounds of communication between the parties, where each round consists of each party exchanging messages with the other parties.

The steps of the online phase are as follow:

1. Each party shares its encrypted input with the other parties using additive sharing scheme, where each share is given to a different party.
2. The parties perform local computations on their shares to compute intermediate values that are required to compute the final output. These computations are performed using arithmetic operations on the shares, such as addition and multiplication.
3. The parties communicate the intermediate values they have computed with each other using secure channels. These intermediate values are encrypted using homomorphic encryption techniques to ensure that they are kept private.
4. Once all intermediate values have been communicated, the parties use the shared intermediate values to compute the final output. This is done using the secret sharing scheme, where each party reconstructs their share of the final output using the shared intermediate values.
5. The parties verify the correctness of the output to ensure that the computation was performed correctly. This can be done by comparing the reconstructed share with the shares received from other

parties. If the reconstructed share does not match the expected value based on the other shares, an error is detected.

Regarding the correctness of the protocol, we know that because n-1 of the parties can be corrupted, we may experience wrong computation of the function or data leakage. We define z as the output of the computation, when parties can learn this output z by sending their shares after the local computations z_i to each other such that everyone can recompute z. However, if some of the participants are dishonest, they can simply lie about their shares, leading to an incorrect result. SPDZ solves this problem by giving everyone, on top of their share z_i, also a share m_i of a MAC m and a share k_i of a key k, such that $\Sigma m_i = m$, $\Sigma k_i = k$, and $m = MAC(k, z)$. When the value z has to be reconstructed, everybody sends their shares of (m, k, z) to everyone else. Now, when assuming that the parties exchange their shares simultaneously, the parties can reconstruct the three values and also check for correctness. This MAC m is both incredibly simple and effective - on a value x under key k it is simply defined as $m = k * x \, mod \, p$ where p is some large prime number. Remember that both the MAC m, the value x and the key k are not known by anyone, since they are secretly shared. This means that we can reuse the same key k over different secret values. So if we have two secret values x_1, x_2 their MACs will be computed as $m_1 = k * x_2 \, mod \, p$ and $m_2 = k * x_2 \, mod \, p$. Then, if each party simply adds their shares of the values and their shares of the MACs, we will get a valid MAC on the resulting value since $(m_1 + m_2) = k * (x_1 + x_2) \, mod \, p$. To achieve active security during the online phase of the secure multiparty computation protocol, it is necessary to ensure that the pre-processing phase generates random Beaver multiplication triples and MACs on all values under a random key k. This ensures that during the computation phase, the MACs will follow the computation due to their homomorphic properties, thus preserving the integrity of the values throughout the computation. Importantly, computing the MACs only requires a multiplication operation between secret values, which means that the same technology used to generate the Beaver multiplication triples in the pre-processing phase can also be used to produce the MACs.

The complexity analysis of the SPDZ protocol involves analyzing the computational and communication costs of the different phases of the protocol. The complexity analysis can be expressed in terms of the following parameters:

- N: The number of parties participating in the computation.
- L: The bit length of the input data.
- l: The bit length of the security parameter used in the protocol.
- m: The size of the arithmetic circuit being computed (number of multiplication gates in the circuit).

The computational and communication complexity of the SPDZ protocol can be expressed as follows:

- Preprocessing Phase: The computational complexity of the preprocessing phase is O($NL^2 + N^2L$), which includes the generation, verification and communication of the precomputed tables.
- Computation Phase: The computational complexity of the computation phase is O(mNL $+m^2L$), which includes the computation of the multiplication gates using the precomputed tables, and the communication of the shares and intermediate values during the computation.
- Share Reconstruction Phase: The computational complexity of the share reconstruction phase is O(NL), which includes the reconstruction of the shares using the secret sharing algorithm, and the communication of the reconstructed shares.

- Error Correction Phase: The computational complexity of the error correction phase is O(mNL), which includes the detection and correction of errors using polynomial interpolation, and the communication of the error correction messages.

In general, it can be observed that the computational complexity of the protocol is contingent upon the parameters specified previously, and it exhibits polynomial growth with respect to them.

BGW Protocol:

The BGW protocol is a cryptographic protocol that enables secure multi-party computation of a function over private inputs from multiple parties. It was introduced in 1988 by Shafi Goldwasser and Silvio Micali, and later improved by Michael Ben-Or and Avi Wigderson. The protocol is based on secret sharing, which is a technique for dividing a secret into multiple shares and distributing them among several parties.

Main Theorem:

- For any n-ary function $f(x_1,..., x_n)$ there exists a protocol for computing f with perfect security in the presence of a **semi-honest adversary** controlling $t < \frac{n}{2}$ parties.
- For any n-ary function $f(x_1,..., x_n)$ there exists a protocol for computing f with perfect security in the presence of a **malicious adversary** controlling $t < \frac{n}{3}$ parties.

The semi-honest case:

Overview of the BGW protocol:

- It is enough to assume that f is a deterministic function.
  - $g(x_1,..., x_n, r)$ can be computed using the deterministic function $f\big((x_1, r_1),..., (x_n, r_n)\big) = g(x_1,..., x_n; \oplus r_i)$.
- We represent f using an arithmetic circuit over a finite field $\mathbb{F}_{(|\mathbb{F}|>n)}$.
  - A circuit where each wire gets a value in F.
  - Gates:
    - Addition gate: $g(a, b) = a + b$ where a and b are values in F.
    - Multiplication with constant gate: $g_c(a) = c \cdot a$.
    - Multiplication gate: $g(a, b) = a \cdot b$.

In secure protocol, each input wire is known to only one party, and that party wants to keep it private. However, when we compute the circuit, we don't reveal any intermediate value throughout the computation. At the end, only the output wires themselves were allowed to be public.

The key idea of BGW protocol is to emulate a computation on a circuit with the inputs of the parties.

Invariant: the value of each wire is hidden using a random polynomial of degree t. It means that the actual value in each wire is going to be a secret shared among the parties.

Shamir's Secret Sharing Scheme:

- $Sharing_{t+1,n}(s)$:
  - Choose a random degree t polynomial with s as its constant term.
  - $p(x) = s + p_1 x + ... + p_t x^t$.
  - Party P_i receives $(\alpha_i, p(\alpha_i))$.
- Properties:
  - Every set of $t + 1$ participants can recover the secret.
  - Every set of t shares doesn't reveal any information about s.

Protocol phases:

1. Input sharing phase:
   a. Each party P_i has a private input x_i, and an assigned value α_i which is known for any other party.
      i. It chooses a random polynomial $g_i(x)$ of degree t for which $g_i(0) = x_i$.
      ii. It sends to each party P_j $(i \neq j)$ the share $g_i(\alpha_j)$.
   b. At the end of this stage, each party P_i holds a share from any other party at its assigned value - $g_1(\alpha_i),..., g_n(\alpha_i)$.
2. Circuit emulation phase:
   a. We will focus on two specific functions:
      i. An addition gate - the inputs for each party will be $g_a(x)$ and $g_b(x)$ (on two different wires), and the goal is to obtain a share on a polynomial of degree t that hides $a + b$ - $h_{a+b}(x)$. The function that we want to compute is

      $$f_{add}\left(\left(g_a(\alpha_1), g_b(\alpha_1)\right),..., \left(g_a(\alpha_n), g_b(\alpha_n)\right)\right) = \left(h_{a+b}(\alpha_1),..., h_{a+b}(\alpha_n)\right).$$

      ii. A multiplication gate - the inputs for each party will be $g_a(x)$ and $g_b(x)$ (on two different wires) and the output will be $h_{a \cdot b}(x)$. The function that we want to compute is $f_{mult}\left(\left(g_a(\alpha_1), g_b(\alpha_1)\right),..., \left(g_a(\alpha_n), g_b(\alpha_n)\right)\right) = \left(h_{a \cdot b}(\alpha_1),..., h_{a \cdot b}(\alpha_n)\right)$.
   b. The computation is emulated using circuit gate by gate - computing shares of the output wire of a gate from the shares of its input wires.
3. Output reconstruction phase:
   a. At this point the parties hold shares of all output wires - each party P_i holds shares $g_{y_1}(\alpha_i),..., g_{y_n}(\alpha_n)$.
   b. $\forall i$, only P_i is supposed to learn y_i.
   c. All parties send their shares $g_{y_j}(\alpha_1),..., g_{y_j}(\alpha_n)$ to P_j and P_j can reconstruct y_j.

f_{add} computation:

- Each party P_i knows $g_a(\alpha_i), g_b(\alpha_i)$.
- The output will be $h_{a+b}(\alpha_i) = g_a(\alpha_i) + g_b(\alpha_i)$ - a polynomial of degree t that hides $a + b$.

- In this case the parties only take the two shares and add them together.
- Constant term - $h_{a+b}(0) = g_a(0) + g_b(0) = a + b$.

f_{mult} computation:

- Each party P_i knows $g_a(\alpha_i), g_b(\alpha_i)$.
- We can't do the same as in f_{add}, because the degree of the polynomial $h(x) = g_a(x) \cdot g_b(x)$ will be $2t$ instead of t (as g_a and g_b are two polynomials of degree t), and h isn't random (h is a composition of two polynomials).
- For any polynomial $h(x)$ with degree $t < n$ there exist constants $\lambda_1,...,\lambda_n$ such that $\lambda_1 \cdot h(\alpha_1) + ... + \lambda_n \cdot h(\alpha_n) = a \cdot b$:

  $$\begin{pmatrix} 1 & \alpha_1 & \alpha_1^2 & \dots & \alpha_1^{2t} \\ 1 & \alpha_2 & \alpha_2^2 & \dots & \alpha_2^{2t} \\ \vdots & & & & \vdots \\ 1 & \alpha_n & \alpha_n^2 & \dots & \alpha_n^{wt} \end{pmatrix} \begin{pmatrix} ab \\ h_1 \\ \vdots \\ h_{2t} \end{pmatrix} = \begin{pmatrix} h(\alpha_1) \\ h(\alpha_2) \\ \vdots \\ h(\alpha_n) \end{pmatrix}$$

  By multiply by the inverse matrix we get:

  $$\begin{pmatrix} ab \\ h_1 \\ \vdots \\ h_{2t} \end{pmatrix} = \begin{pmatrix} \lambda_1 & \dots & \lambda_n \\ \vdots & & \\ \dots & & \end{pmatrix} \begin{pmatrix} h(\alpha_1) \\ h(\alpha_2) \\ \vdots \\ h(\alpha_n) \end{pmatrix}$$

  And as a result, $a \cdot b = \lambda_1 \cdot h(\alpha_1) + ... + \lambda_n \cdot h(\alpha_n)$.

- Each party P_i can compute $h(\alpha_i)$.
- The protocol for party P_i:
  - Compute $h(\alpha_i) = g_a(\alpha_i) \cdot g_b(\alpha_i)$.
  - Share $h(\alpha_i)$ using a degree t polynomial $H_i(x)$.
  - Given all the shares that were received $H_1(\alpha_i),..., H_n(\alpha_i)$, output $\lambda_1 \cdot H_1(\alpha_i) + ... + \lambda_n \cdot H_n(\alpha_i)$.
- Why does it work?
  - The parties compute a share on the polynomial $H(x) = \lambda_1 \cdot H_1(x) + ... + \lambda_n \cdot H_n(x)$, and each P_i outputs $H(\alpha_i)$.
  - This is a polynomial of degree t, because each one of $H_1(x),..., H_n(x)$ is of degree t.
  - It is random because each one of $H_1(x),..., H_n(x)$ is random.
  - Its constant term is ab:

    $$H(0) = \lambda_1 \cdot H_1(0) + ... + \lambda_n \cdot H_n(x) = \lambda_1 \cdot h(\alpha_1) + ... + \lambda_n \cdot h(\alpha_n) = a \cdot b.$$

- In conclusion, it's a perfectly secure protocol for a semi-honest adversary in which the number of corrupted parties is less then $\frac{n}{2}$. We used it in the multiplication gate, as otherwise we don't have enough information.

Security:

- Input sharing phase - t shares only polynomials of honest parties.
- Circuit emulation phase - in addition we don't do anything, and in each multiplication, the adversary receives t shares on each one of the polynomials $H_1(x)$,..., $H_n(x)$ which hides the values.
- Output reconstruction phase - given the t shares on the output wires of the corrupted parties to the simulator as an input - the simulator fills the missing points of the polynomials, reconstructs the polynomial and sends the remaining shares.

The malicious case:

Security:

- The parties jointly compute $f(x_1,..., x_n)$:
  - The honest parties provide **true** inputs.
  - The corrupted parties might provide **any input** they like. If they don't cooperate, the honest parties can choose default inputs for them.
- Privacy - the adversary doesn't learn any information on the honest parties' inputs.
- Guaranteed output delivery - the adversary cannot prevent the honest parties from obtaining their outputs.

What might go wrong?

Input sharing phase

A corrupted party might send shares that don't lie on a polynomial of degree t.

Circuit emulation phase

During the addition gates, nothing can go wrong because there isn't an interaction.

During the multiplication gates, if we're looking at the multiplication gate of the semi-honest case, the adversary can send a polynomial that isn't of degree t (we can deal with it using VSS), and the secret that we input to the VSS might be wrong (this is the actual problem of the multiplication protocol). In order to fix this, we will add a verification before sending the values to VSS.

Simplified case - $t < \frac{n}{4}$:

Let $h(x) = g_a(x) \cdot g_b(x)$, where h is a polynomial of degree $2t$. Each party computes a share on this polynomial by just computing $h(\alpha_i) = g_a(\alpha_i) \cdot g_b(\alpha_i)$. Reed Solomon code is $(n, k+1, n-k)$-code (which means that we can take a message of size $k+1$, increase it to size n, get some redundancy and the distance between two code words is $n-k$) and can correct $\frac{n-k-1}{2}$ errors. When $n = 3t+1$, for $k = 2t$ we have a

$(3t + 1, 2t + 1, t + 1)$-code which can correct $\frac{t}{2}$, and when $n = 4t + 1$, for $k = 2t$ we have $(4t + 1, 2t + 1, 2t + 1)$-code which can correct t errors.

Facts from error correcting code - Let $C \subset \Sigma^n$ be a (n, k, d)-linear code. A generator matrix $G \in \Sigma^{k \times n}$ is a matrix that maps "messages" into codewords. For $m \in \Sigma^k$ we have that $m \cdot G \in \mathbb{F}^n$ is a codeword. A parity check matrix $H \in \Sigma^{(n-k) \times n}$ is a matrix that satisfies $G \cdot H^T = o^{k \times (n-k)}$, which means that for every codeword $c \in C$ (i.e., there exists some $m \in \Sigma^k$ such that $m \cdot G = c$) $c \cdot H^T = 0$, and for every "noise" codeword $\widetilde{c} = c + e \in \Sigma^n$ where $c \in C$ and $e \in \Sigma^n$ is of distance 0 - $\widetilde{c} \cdot H^T = (c + e) \cdot H^T = c \cdot H^T + e \cdot H^T = e \cdot H^T$, it's possible to find e from $e \cdot H^T$ and $e \cdot H^T$ doesn't contain any information about m.

When $n = 4t + 1$, each party computes $h(\alpha_i) = g_a(\alpha_i) \cdot g_b(\alpha_i)$ and sub-shares it. Let $\widetilde{c} = c + e$ where $c = \left(h(\alpha_1),..., h(\alpha_n)\right)$ and the distance of e from 0 is at most t. We can run a check - if some P_i inputs something wrong, we want to identify it and to correct it, i.e. the honest parties will change their sub-shares of P_i to $h(\alpha_i)$. The check is that each party P_i sub-share its input using some $H_i(x)$, where $H_i(x)$ hides $h(\alpha_i)$. Parties compute the "circuit" $\widetilde{c} \cdot H^T$ and reconstruct $e = (e_1,..., e_n)$. The parties can see if there are errors, where are the errors and what are the errors. For every $e \neq 0$, reconstruct $H_i(0)$ and "correct" the sub-share to $H_i(0)$ - e_i.

In conclusion, each party holds $g_a(\alpha_i), g_b(\alpha_i)$, multiplies $h(\alpha_i) = g_a(\alpha_i) \cdot g_b(\alpha_i)$, sub-share $h(\alpha_i)$ and check and correct wrong inputs. Now each party P_j holds a share on one of the polynomials $H_1(x),..., H_n(x)$ that hide $h(\alpha_1),..., h(\alpha_n)$ respectively. That is, P_j holds $H_1(\alpha_j),..., H_n(\alpha_j)$, and the output will be $\lambda_1 \cdot H_1(\alpha_j) +... + \lambda_n \cdot H_n(\alpha_j)$.

When $n = 3t + 1$, we don't check and correct wrong inputs because in this case we can correct only $\frac{t}{2}$ errors for a polynomial of degree $2t$. Instead, the process will be as described:

- Input - each party holds $g_a(\alpha_i), g_b(\alpha_i)$.
- Each party sub-shares $g_a(\alpha_i)$ and $g_b(\alpha_i)$. Since $g_a(\alpha_i), g_b(\alpha_i)$ are of degree t, we can guarantee that the right values were shared.
- Each party sub-shares $h(\alpha_i) = g_a(\alpha_i) \cdot g_b(\alpha_i)$ and "proves" that those sub-shares agree with the sub-shares of $g_a(x), g_b(x)$.
- Now each party P_j holds a share on one of the polynomials $H_1(x),..., H_n(x)$ that hide $h(\alpha_1),..., h(\alpha_n)$ respectively. That is, P_j holds $H_1(\alpha_j),..., H_n(\alpha_j)$.
- The output will be $\lambda_1 \cdot H_1(\alpha_j) +... + \lambda_n \cdot H_n(\alpha_j)$.

Output reconstruction phase

Parties might send wrong shares instead of sharing the actual value that they have.

We can fix it using VSS. Let $t < \frac{n}{3}$. There exists a perfectly secure **Verifiable Secret Sharing** protocol in the presence of a malicious adversary.

- Privacy - for an honest party, the adversary learns nothing about s.
- Consistency - the outputs of the honest party are consistent with some s^* even if the adversary is corrupted.
- Correctness - for an honest party, consistency holds with $s^* = s$.

In the reconstruction phase, even if corrupted parties send wrong shares, honest parties can still recover the secret.

Code

Yao's Protocol Implementation:

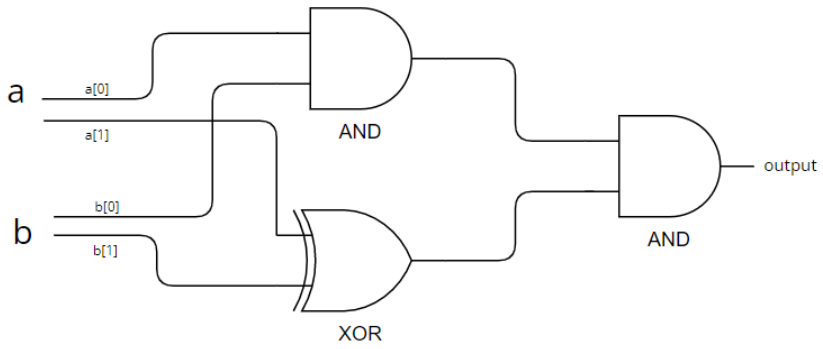
The problem space, which has already been introduced before, is to compute some joint function over two parties' input, without each receiving any information about the input of the other party during the process.

Motivation:

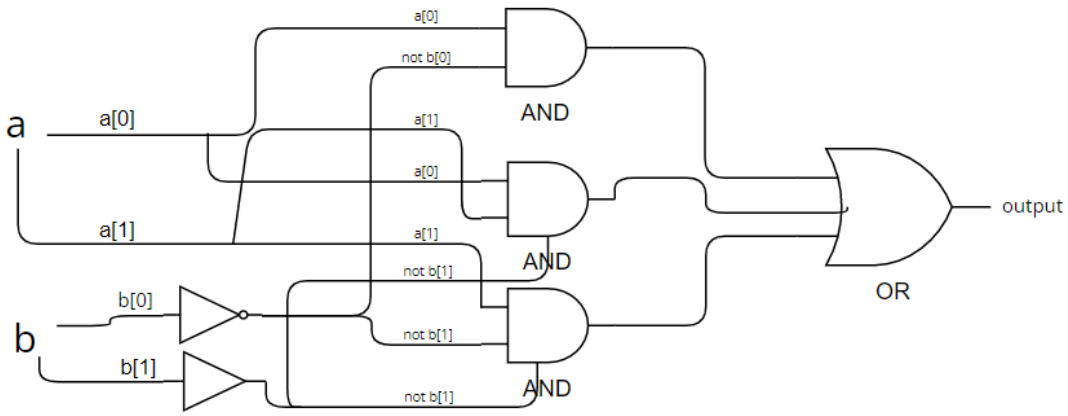
There are multiple motivations to use such a protocol. One common use case is when two parties wish to jointly perform a computation on sensitive data, such as medical records or financial information, but do not want to disclose that data to each other or to any third parties. In such cases, 2-party computation can provide a way to perform the computation without compromising the privacy of the data. Another use case is when two parties wish to collaborate on a task, such as a scientific experiment or a business negotiation, but do not fully trust each other. By using 2-party computation, the parties can jointly perform the task while ensuring that neither party can cheat or manipulate the results.

In our example, we constructed 2 boolean circuits that we wish to compute:

Circuit 1:



Circuit 2:



The first circuit is some basic example for a circuit we are able to compute securely, and the second one computes the results whether the input of the first party is greater than the input of the second party, which is is a partial solution to the Millionaires Problem introduced by Yao and was shown in the previous pages. As we already discussed earlier, we can compute any function using such boolean gates, and so this protocol's implementation shows examples which can easily be modified for more complex ones. Moreover, we used 2 different circuits, each with a different number of gates, in order to see the difference in the running time as a function of the number of gates used.

Architecture:

First we'll mention that all of the terms being used here are fully explained in the presentation of Yao's protocol.

Our solution is composed of 5 classes:

1. **Garbler** - Party A, responsible for garbling the circuit and forwarding the garbled circuit to the evaluator. It holds a few random variables: Its input (2 bits), keys (2 for each wire), signal bits (1 for each wire) and lambda values (1 for each wire). In the process of garbling the circuit, it sets a key for each internal value of a wire (according to the lambda value of this wire which can flip the order), sets each wire belong to the input of Party A its input with the key corresponds to his value, and also sets an external value for that wire according to the signal bit of the wire and its internal value. The garbler also creates a table for each gate (according to the order of the external values) for decrypting the key of the output wire, and a final table only for receiving the final output of the circuit. Moreover, the garbler implements a mechanism of oblivious transfer, in order to send the evaluator the corresponding keys for his inputs without getting any information about them.
2. **Evaluator** - Party B, responsible for evaluating the garbled circuit and returning the final result of the computation. He holds his private input (2 bits), and performs the evaluation in the following steps: Activates all OTs in parallel and receives all the suitable keys and signal bits according to his inputs, takes the correct cell in the gate's table according to the external values of the wires, and decrypt this cell with wires' keys to receive the key of the output and its external value. For the last gate, it takes the output key and reveals the final output value by using the final table after the decryption.
3. **Wire** - Each wire object holds multiple public variables: Its serial number, keys (1 for each value) and its external value.
4. **Gate** - Each gate object holds multiple public variables: Gate type, 2 input wires objects and 1 output wire object, a table and an final output table (only if this is the last gate in the circuit).
5. **Circuit** - Each circuit object holds all its gates and the serial number of the last wire in the circuit.

Oblivious Transfer

We decided to implement an oblivious transfer protocol that was suggested by Tung Chou and Claudio Orlandi and is based on the Diffie-Hellman Key Exchange that we studied in class. In our implementation, the sender will be the Garbler and the receiver will be the Evaluator.

The protocol works as followed:

1. We randomize two numbers - **g**, that will be public to both the sender and the receiver, and **a** that will be used explicitly only by the sender.
2. The sender calculates the value of $A = g^a$, and sends it to the receiver, in addition to g (the value of g is randomized in the Garbler class as part of the oblivious transfer function, but both the Garbler and Evaluator are using it).
3. The receiver randomizes a number **b**, and calculates the the value of B according to the chosen-bit, as following:
    a. If the chosen-bit is equal to 0, then $B = g^b$.
    b. If the chosen-bit is equal to 1, then $B = A \cdot g^b$.

    In addition, at this point the receiver calculates its decryption key - $k = H(A^b)$, where H is the sha256 hash function. The receiver returns to the sender the value of B.

4. The sender now computes the value of two keys:
    a. $k_0 = H(B^a)$.
    b. $k_1 = H\left(\left(\frac{B}{A}\right)^a\right)$.

    In both cases H is the sha256 hash function. At the end of this stage, one of those keys, according to the chosen-bit (which the sender doesn't know), will be equal to the value of k that was calculated by the receiver.

5. The sender encrypts the messages (in our case, the keys for the input wires) with the keys that were created last, and will return the encrypted messages in addition to the signal bits.
6. The receiver now will be able to decrypt the requested message, according to the chosen-bit, without the sender knowing which message is decrypted.

Note: In order to prevent cases of overflows (and as we raise g to the power of a and then to the power of b), a, b will be randomized in the range of 0 to 5 and g will be randomized in the range of 1 to 5 (to prevent division in 0).

The Attacker Model:

The attacker model in this protocol is that one of the parties may be a semi-honest adversary, who follows the protocol correctly but tries to learn more information than they are supposed to by observing the messages sent during the protocol execution. This design good for against such attacker by using the following techniques:

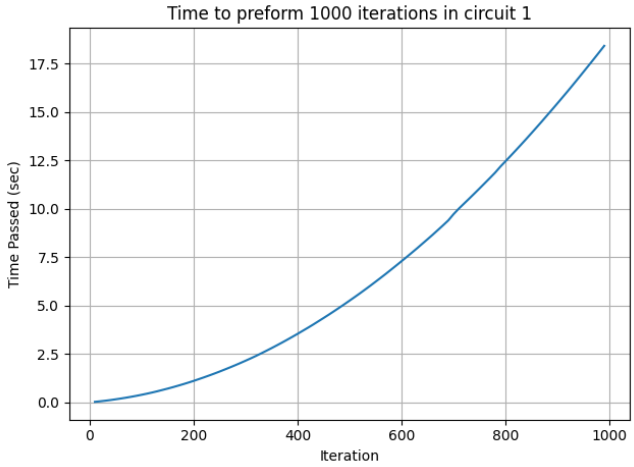
- **Oblivious transfer**: The protocol uses oblivious transfer (OT) to enable Party B send its input to Party A and receive the corresponding key without revealing the input itself. OT ensures that the receiver only learns one of the sender's values, chosen by the receiver.
- **Garbled circuits**: The protocol uses garbled circuits to compute the function securely. We use that technique for encoding the function in a way that hides all inputs and intermediate outputs of the function's computation, while each party can view only their inputs and the final output, and all those intermediate values of the computation are seen only as keys that were generated randomly and doesn't reveal any secret information.

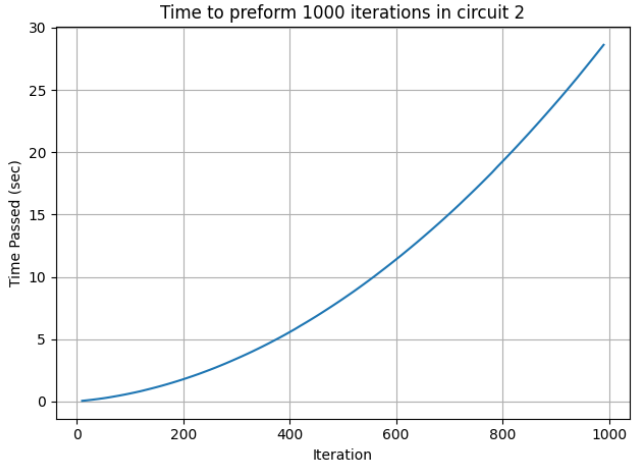
- **Randomization**: The protocol introduces randomization into the computation by using the lambda variables, so that each value gets a key in a fully random way. That technique makes it difficult for a semi-honest adversary to deduce information about the inputs or the function being computed.

Evaluate the solution:

Overall, the solution for Yao's protocol is considered to be a highly effective and efficient solution for secure two-party computation, especially against semi-honest adversaries. One of the key strengths of the Yao protocol is its ability to securely compute any function that can be expressed as a Boolean circuit. This makes the protocol very versatile and widely applicable, as many real-world applications can be expressed in this form. Additionally, the protocol's use of garbled circuits allows it to achieve a high level of security by hiding the inputs and outputs of the function being computed. Another strength of such implementation is its efficiency. While secure two-party computation is inherently more computationally expensive than non-secure computation, this protocol has been shown to be highly efficient in practice, especially when compared to other secure computation techniques. One potential weakness of the Yao protocol is its vulnerability to malicious adversaries, who may try to actively disrupt or subvert the protocol in order to learn more information than they are supposed to. Overall, while it may not be the best choice in all scenarios, it is a well-established and widely trusted protocol.

To investigate the variance in the execution time of the protocol across circuits of varying gate counts, we conducted an experiment wherein we evaluated the performance of two implemented circuits consisting of 3 and 7 gates, respectively. Each circuit was run for 1,000 iterations, and the overall time elapsed was recorded. The obtained results of this experiment are presented below:





BGW Protocol Implementation:

Motivation:

As described above, the BGW (Ben-Or, Goldwasser and Wigdersion) protocol is a cryptographic protocol that enables secure multi-party computation of a function over private inputs from multiple parties. This protocol can be used to evaluate an arithmetic circuit over a finite field $\mathbb{F}$, consisting of addition, multiplication and multiplication by constant gates. The protocol is based on Shamir secret sharing, which under this theorem the shares are homomorphic in a special way - the underlying shared value can be manipulated obliviously by suitable manipulations to the individual shares.

Architecture:

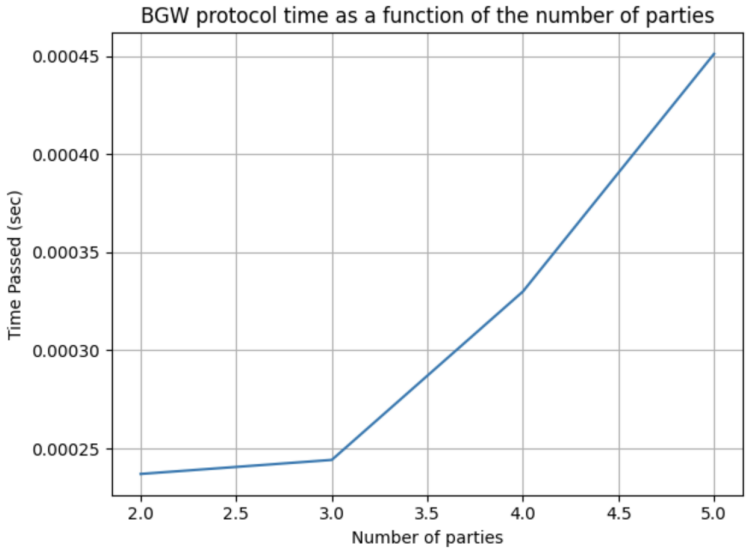
At first, we will describe the Party class:

- In the initialization, the class gets a randomized prime number and the polynomial degree *t*.
  - Each party randomizes a secret value (between 0 to $p - 1$), coefficients for its polynomial and an assigned value.
  - It calculates the polynomial on the assigned value and adds it to an array which will hold all the parties' shares.
- The class has a few functions:
  - share_value function - the function gets another party and its assigned value, and sends the polynomial in the other party's assigned value, as part of the input sharing phase.
  - add_share_from_party function - the function gets a share from another party and adds it to the shares array.
  - get_assigned_value function - the function returns the party's assigned value, i.e. for each P_j it will return α_j.
  - get_addition_share function - the function returns the sum of all the shares (as part of evaluating the addition function in the reconstruction phase).
  - reconstruct_share_sum function - the function gets an array of tuples that represent each other parities' assigned value and the value of f_{add} at this value. it computes the value of $f_{add}(0)$ without revealing each party's secret (as we know that $f_{add}(0) = h_1(0) + ... + h_n(0) = x_1 + ... + x_n$, where x_i is P_i's secret). We are doing it using the tuples from the circuit emulation phase - $(\alpha_i, f_{add}(\alpha_i))$, and using Lagrange Interpolation formula (which allows us to reconstruct f_{add} polynomials using the values we have):
    - $\forall 1 \le i \le n,\ l_i = \frac{x-x_0}{x_i-x_0} \cdot ... \cdot \frac{x-x_{i-1}}{x_i-x_{i-1}} \cdot \frac{x-x_{i+1}}{x_i-x_{i+1}} \cdot ... \cdot \frac{x-x_{n-1}}{x_i-x_{n-1}},\ f_{add}(x) = \sum_{i=0}^{n-1} y_i l_i(x)$, where $y_i = f_{add}(\alpha_i)$.
    - Instead of reconstructing the polynomial and than compute it on 0, we set $x = 0$ when defining l_i and at the end of this phase we get $f_{add}(0) = f_{add}(0) = h_1(0) + ... + h_n(0) = x_1 + ... + x_n$.
  - print_secret function - a function that prints the party's secret. We're using this function only in the end, when we want to verify that the addition function's result is equal to the sum of the parties' secrets.

We will now describe the main function:

- In order to examine the running time of the algorithm as a function of the number of parties, we're going to check its performance from 2 parties to 5 parties. Because we want in the reconstruction phase to evaluate the exact value of $f_{add}(0)$ (as described earlier and will be describe) using Lagrange Interpolation, we have to set the degree of the polynomial to be $n-1$, where n is the number of parties.
- In addition, we randomize a prime number p which is smaller than 500, and set the maximum value of a coefficient in each polynomial and of the assigned values to be 15 (both are arbitrary values which have been chosen in order to prevent overflows).
- We initialize each party as described earlier, and create an array containing all the parties.
- Input sharing phase - in this phase each party sends to each other party the value of its polynomial on the other party's assigned value. I.e., each party P_i send to each other party P_j $(i \neq j)$ the value of $h_i(\alpha_j)$, where h_i is the polynomial we described above and α_j is P_j's assigned value.
- Circuit emulation phase - at the end of the input sharing phase, each party P_i hold the shares $h_1(\alpha_i)$,..., $h_n(\alpha_i)$. At this phase, we compute for each party the value of $f_{add}(\alpha_i) = h_1(\alpha_i) + \ldots + h_n(\alpha_i)$. At the end of this phase, we have the tuple $(\alpha_i, f_{add}(\alpha_i))\ \forall 1 \leq i \leq n$.
- Output reconstruction phase - on this phase we want to compute the value of $f_{add}(0)$ without revealing each party's secret, using the tuples from the circuit emulation phase - $(\alpha_i, f_{add}(\alpha_i))$. As we don't want a third-party to do the computation, we randomize a party out of the parties we created earlier, and it will do the computation as described above. We notice that as the party only gets the value of $(\alpha_i, f_{add}(\alpha_i))\ \forall i$, it can't learn anything about each other parties' secret, as it doesn't have any information about each other party apart from its assigned value (which it gets already as part of the input sharing phase) and the value of $f_{add}(\alpha_i)$.

As described above, in this program we examined the running time of the algorithm as a function of the number of parties, we're going to check its performance from 2 parties to 6 parties:



As can be seen from the graph and as expected, the time that takes for the BGW protocol is growing as the number of parties gets higher, which makes sense as we have more variables that should be taken into account and more computations to do.

In conclusion, at any point each party P_i knows only the shares that it got on the input sharing phase - $h_1(\alpha_i)$,..., $h_n(\alpha_i)$, and it can't learn any other party's secret or polynomial. Both were set as private fields, which means that we cannot access them outside the class itself. In addition, at the end of the process we are able to calculate the value of $f_{add}(0) = x_1 + ... + x_n$ where x_i is P_i's secret, without knowing the secrets themselves.

Note: in this code we tried to avoid situations of an overflow. However, if an overflow did occur, an exception with a corresponding message will be raised.

Conclusion

At this paper we examined the usage of Multi-Party Computation (or MPC), a cryptography subfield that focuses on different computational methods that aims to enable different parties to interact and compute a joint function.

In particular, we went through the settings and definitions of MPC, and took a closer look at few of the protocols that exists today:

- **Yao's 2PC Protocol and BMR Protocol** - Yao's Protocol was the first protocol for secure computation, which initially aimed for the two party case that wants to compute some function securely. The protocol works by having one party (the garbler) encode the function as a garbled circuit and send it to the other party (the evaluator), who evaluates the circuit on their private input. The garbled circuit ensures that the evaluator does not learn anything about the garbler's input and the garbler does not learn anything about the function or the evaluator's input. The improvement of BMR was to solve this problem for the case of more than two parties.
- **GMW Protocol** - this protocol is a classic multi-party computation (MPC) protocol proposed by Goldreich, Micali, and Wigderson (GMW). The basic version of the protocol, which assumes semi-honest behavior, uses a boolean-circuit representation for the computation, and is based heavily on Oblivious Transfer to execute the boolean gates. This protocol is less secure but more secure than BMR, and is also widely used in practical applications due to its simplicity.
- **SPDZ Protocol** - a protocol in the dishonest majority section of secure multiparty computation (MPC), which efficiently utilizes a form of homomorphic encryption. The protocol is a two-phase protocol where inputs are shared via an additive secret sharing scheme. The offline phase allows parties to precompute and exchange certain values to reduce the computational overhead of the online phase. SPDZ is efficient and has been used in several real-world applications, including secure machine learning and privacy-preserving auctions.
- **BGW Protocol** - a cryptographic protocol that enables secure multi-party computation of a function over private inputs from multiple parties. It was introduced in 1988 by Shafi Goldwasser and Silvio Micali, and later improved by Michael Ben-Or and Avi Wigderson. The protocol is based on secret sharing, which is a technique for dividing a secret into multiple shares and distributing them among several parties.

In the implementation phase of the project, we implemented two of the above protocols - Yao's Protocol and BGW protocol, and examined different situations of each.

In addition to the advantages of using MPC, it also has a few disadvantages which need to be taken in consideration:

1. High computational and communication costs - the techniques used in MPC require significant computational overhead and communication costs compared to traditional computation, which can be a bottleneck in large-scale distributed systems. This is because the parties need to exchange multiple rounds of communication and perform complex computations to secure their inputs and compute the final output.
2. Limited applicability - MPC protocols are designed for specific types of computations, which may not be suitable to all possible types of applications.

3. Potential of collusion - Although MPC is designed to prevent any party from learning about the others' private inputs, there is still the potential for collusion among some or all of the parties. This can result in a breach of privacy and compromise the security of the system.

In conclusion, in the last few years MPC has changed the way we think about sharing information and has opened the doors to a wide variety of privacy based on products and services. In addition, it has provided techniques that allow us to draw conclusions without worrying about safety and privacy.

Bibliography

- Secure Multiparty Computation: Introduction, by Ran Cohen (Tel Aviv University)
- Secure Multiparty Computation (MPC) by Yehuda Lindell
- Secure Multiparty Computation Goes Live
- The Yao and BMR Protocols for Secure Computation by Yehuda Lindell, Bar-Ilan University
- The GMW and BMR Multi-Party Protocols by Benny Pinkas, Bar-Ilan University
- Secure multi-party computation (GMW Protocol + Malicious Model)
- MPC Protocols - MPC Wiki
- The SPDZ Protocol by Prof. Ivan Damgård
- Multiparty Computation from Somewhat Homomorphic Encryption
- Homomorphic Encryption in the SPDZ Protocol for MPC by Peter Scholl
- MPC Techniques Series, Part 9: SPDZ
- SPDZ Protocol - Secret Sharing Presentation
- Fundamental MPC Protocols
- Shamir's Secret Sharing Algorithm
- Secure Multi Party Computation part 1- The BGW Protocol - Gilad Asharov
- Secure Multi Party Computation - The BGW Protocol
- What is Multi-Party Computation (MPC)?
- The Simplest Protocol for Oblivious Transfer