
2017-2018

# COMP 4805

## Blockchain Research & Implementation

Yip Lai Ting
3035193428

# Content

# A. Abstract

Secure multi-party computation (MPC) allows multiple parties to jointly compute a function over their inputs while keeping them private. Oftentimes, it is difficult to maintain a consensus over the correct state of information, as well as identifying the dishonest party. The blockchain technology provides a secure platform for MPC as it keeps track of all operations and the change of state information brought by the operations cannot be maliciously reverted. Message exchange and identification of dishonest parties can hence be done in a more efficient manner. The project attempts to realize MPC on a popular blockchain platform, known as Ethereum.

# B. Background

### B.1 What is Ethereum?

Recall that Bitcoin stores the transaction of coins in the blockchain. Therefore, the state information is only the amount of coins in each account, e.g. Account A has 5 coins, Account B has 10 coins, etc. What if we want to make the whole network a computer for general purpose computations and no one can maliciously modify the program states? Ethereum does this.

For example, if we deploy a voting program on Ethereum, through the blockchain's broadcast, everyone will then get a copy of this program. Now, suppose a person casts a vote by calling the "vote" function of this program, this will modify his voting status from "not yet voted" to "voted". Through the blockchain's broadcast, others will update the state information of this voting program. The state information here refers to the voting status of this person. Blockchain ensures that no malicious modifications could be done. In this situation, we are assured that this person's voting status cannot be maliciously modified back as "not yet voted".

### B.2 What is Gas?

Gas is the execution costs for every operation made on the Ethereum blockchain. Each time we execute a program's function, it will consume gases and we have to pay coins (Ether) for it. The amount of Ether to pay is calculated by the amount of consumed gases times the gas price at that moment.

### B.3 What is Solidity Smart Contract?

You can imagine a smart contract is a program written in Solidity (.sol). Apart from it, a smart contract is also an account. There are two types of account in Ethereum. The first type is the one controlled by humans, known as the external account. If you know the account's address – the public key, you can send Ether to it. The other type is the contract account. When a person deploys a smart contract, he will get back the contract account's address. Just as external accounts, you can transfer Ether to contract accounts. The difference is that a contract account has codes and state information.

In Ethereum, everyone can deploy new contracts, and call functions of other contracts as long as he knows the contract account's address.

**B.4 What is Ethereum Node?**

It is simply a computer which participates in the Ethereum network by keeping a copy of the blockchain, usually as well as verifying the change of state information (mining).

**B.5 What is MetaMask?**

To modify a contract's state information, you have to execute the contract's function through any Ethereum node. It could be on your local machine or elsewhere. Your contract can have a web front-end. However, the regular browser does not know how to interact with an Ethereum node, and so this is where MetaMask comes in. It is a Chrome plugin that enables the webpages to send execution requests to an Ethereum node.

What is more is MetaMask allows us to specify which Ethereum node to send these requests to. The ability to send requests to nodes outside of our local machine is crucial because it implies that we do not have to run an Ethereum node by ourselves. Commonly, MetaMask can route our requests to Infura which maintains a myriad of public Ethereum nodes.

**B.6 Why Secure Multi-Party Computation (MPC) on Blockchain?**

MPC allows multiple parties to jointly compute a function over their inputs while keeping them private. As a piece of data is split into several shares, even a simple multiplication can be inefficient, e.g. every party has to exchange their intermediary results to each other in order to further compute the final result (this process is known as degree reduction), thereby resulting in a considerable amount of communication latency. Besides, the requirement of trust is onerous which it is difficult to tell if a party has done the computation properly, and consequently difficult to punish dishonest parties.

Given that a block in the blockchain encapsulates all the valid changes of state information, it can represent as a synchronized global clock for message exchange. Additionally, the function execution is recorded and its result cannot be maliciously modified, hence it helps to discourage dishonest behaviors. Blockchain's relative efficiency in achieving consensus establishes a secure environment for MPC.

Furthermore, recall that everyone can call functions of other contracts on Ethereum, it indicates an opportunity to outsource computational power. The person who asks for the result does not need to be the one who performs the computation. Instead, he can pay a fee and let others to perform the computation. As a result, it improves the scalability of MPC in a sense that we can request result without worrying about the computational resources.

## C. Objective

Zyskind (2016) proposes an MPC framework built on blockchain. The goal of the project is to realize some of the concepts in this framework on Ethereum. The focus here is how the contract (program) functions rather than what it actually computes, therefore I chose a simple type of computation – linear regression analysis. Yet, the mechanisms described in this project such as when and how to punish dishonest parties, is believed to be applicable to other types of computation as well.

## D. System Overview

### D.1 Role

There are 4 roles in the contract. The first role is the "Initiator" who manages the computation. The second role is the "Data Storer" who stores shares. The third role is the "Requester" who submits computation requests. The last role is the "Computation Party" who performs computation. These roles are not mutually exclusive, for instance, a data storer can also be a requester.

### D.2 Mechanism

Any Ethereum node can register to be a computation party if he sends enough Ether to the contract's account, known as a deposit. The rationality is that if this party later fails to perform the computation or being dishonest, the contract can punish him by deducting his deposits.

The minimum amount of deposits is determined by the initiator. To avoid having too many computation parties, the initiator is also responsible for setting a ceiling for the number of computation party. For example, if the ceiling is 3 and there have already been 3 computation parties, then for any subsequent registrations, we will still collect their deposits. However, these nodes will be put into the pending list. They will be promoted to be a computation party when the existing computation parties withdraw or get disqualified.

A person cannot register to be a data storer or a requester. It is the initiator's responsibility to recognize the data storer or the requester by adding this person's address into the data storer's list or the requester's list. Only after that, this person will have the right to call the data storer's functions or the requester's functions. In other words, if an unrecognized Ethereum node attempts to store shares or submits requests, the contract will reject it.

A data storer will not store the original data. He will split the data into several shares and store the shares. Each share will be distributed randomly to a computation party, so later the party can perform computation based on the shares he has received. For security reason, the contract will only allow the data storer to store shares when the number of computation party is at least equal to the number of share he is going to store. The reason is that imagine there is only 1 computation party and 3 shares split from the same data. No

matter how random the algorithm is, the 3 shares will still be distributed to this particular computation party. Under this circumstance, the party can easily construct back the original data, indicating that the data is no longer private. It is thus necessary to ensure the number of computation party is sufficient when the data store stores shares.

A requester can submit a computation request associated with a fee to the contract. The fee will later be evenly distributed to the computation parties as a reward for their computation work. The fee is kept by the contract until the computation has completed. The initiator has the responsibility to determine the minimum amount of computation fee that a requester has to submit, as well as the maximum number of request that a requester can submit in a round. This is to ensure the computation party will receive a reasonable amount of reward for his work. It can also make sure that there will not be too many requests in a round overloading the computation parties.

Lastly, it is the initiator's responsibility to specify the duration of the computation period. He is also responsible for declaring the start of the computation period. Only after that, the computation parties can perform computation. Other operations are forbidden, e.g. request submission, data storage, etc. during the computation period.

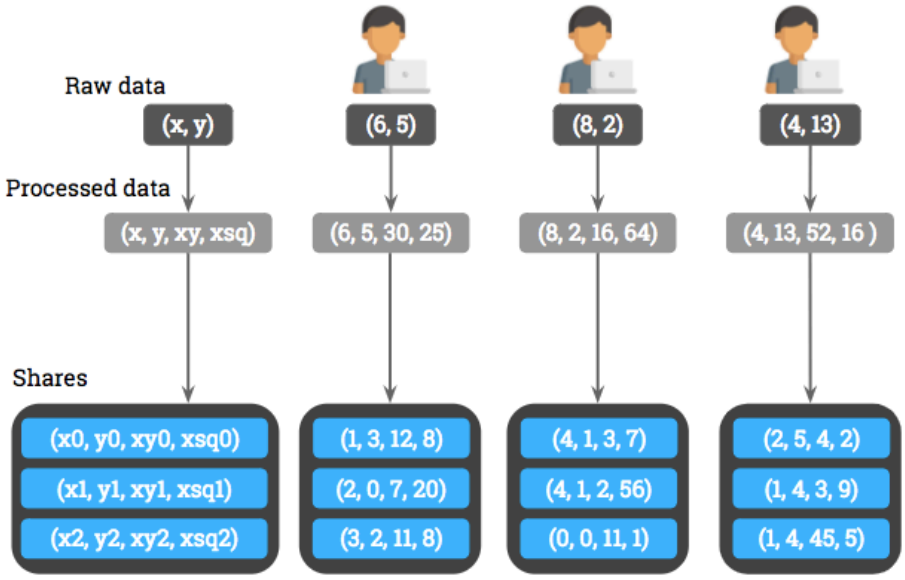
The end of the computation period marks the beginning of the checking period. It is the buffering time for the requester to check the result. During this period, computation and other operations are disallowed. Similarly, it's the initiator's responsibility to specify the duration of the checking period. He is also responsible for officially declaring the end of the checking period. At this moment, all the requests and results will be cleared, and more importantly, rewards and penalties will be distributed and collected respectively. Upon settlement, other operations are resumed back, e.g. the data storer can store shares, the requester can submit requests, etc.

For a complete rundown, please visit https://www.youtube.com/watch?v=3kErla3aHQA which is a video demonstrating from the start of contract instantiation to the end of checking period via a web front-end. For the source codes, please visit https://github.com/reminiyip/fyp_revised

## E. Data Storer

### E.1 Store Shares

The regression data can be represented as (x, y). For this project, we assume the data storer will process the data by calculating xy and x square, and then split the data – (x, y, xy, xsq) into several shares. Suppose the contract requires a piece of data to be split into 3 shares (split size = 3). The 1st data storer below can hence call the store functions to store (1, 3, 12, 8), (2, 0, 9, 20) and (3, 2, 11, 8).




When calling the store function, besides the shares, the data storer should also provide a date, indicating when these shares (this data) becomes valid, say "2017-01-01". Then, the contract will verify 4 things,

1) whether the caller is a data storer
2) whether the number of computation party is sufficient, which is equal or greater than (>=) the split size;
3) whether the number of shares submitted is equal to (=) the split size
4) whether the date is valid, e.g. "2017-02-31" is an invalid date

Recall the extreme example in Section D which the split size is 3 and there is only 1 computation party. No matter how random the algorithm is, all 3 shares will be distributed to this particular computation party. Hence, the 2nd condition is to prevent such situations.

If all the above conditions are satisfied, the shares will randomly be distributed to the computation parties. One thing worth nothing here is that the contract only guarantees if a party is selected hold a share, he will not hold the next share. Therefore, in this case, if a computation party holds (1, 3, 12, 8), he will not hold (2, 0, 7, 20). However, he can hold (3, 2, 11, 8).

This mechanism is secure because from the perspective of the computation party, he cannot tell which share is split from which data. For example, even if a computation party holds (1, 3, 12, 8) and (3, 2, 11, 8), he will not know if these two shares are split from the same data.

Last but not least, the contract will keep track of the number of data stored and the number of shares stored. Each time a data storer calls the store function successfully, this 2 pieces of state information will be updated accordingly.

## F. Requester

### F.1 Submit Request

The requester can submit a computation request with a fee. He should provide a unique ID for identification purpose and the range of data he wants, e.g. from 2017-01-01 to 2017-12-31. The contract will verify 5 things when a requester calls the request function,

1) whether the caller is a requester
2) whether the request ID is unique
3) whether the date range is valid, e.g. "from 2017-01-01 to 2016-12-31" is an invalid range
4) whether the number of request from this requester is smaller than (<) the maximum no. of request per requester
5) whether the fee is greater than or equal to (>=) the minimum amount of computation fee

For example, if the maximum no. of request per requester is 2 and a requester has already submitted 2 requests, then the 3rd request he submits will be rejected by the contract. Or, if the minimum amount of computation fee is $20, but a requester submits a request with a fee of $10, this will be rejected as well.

One thing worth noting here is that when a requester calls the request function, the contract can get his address by inspecting the caller's address. The caller here refers to the one who calls the request function. Therefore, besides the ID, the date range and the fee, each request also includes the requester's address, despite the requester does not provide his address as an argument when he calls the function.

### F.2 Cancel Request

Given that each request can be identified with a unique ID, a requester can cancel his request by providing the ID. One may ask if another requester can cancel a request not submitted by him? It is impossible to do so. Recall that each request includes the requester's address. When a person calls the cancel function, the contract will first retrieve the request based on the ID. Then, it will check the requester's address against the caller's address. The caller here refers to the one who calls the cancel function. The cancellation will be rejected if no request can be found based on the ID or the address does not match.

### F.3 Check Result

A requester can check his request's result by providing the ID. Similar to cancellation, it is impossible for another requester to check a request's result if the request was not submitted by him. This is because the contract will first retrieve the request based on the ID and then verify the requester's address against the caller's address. The caller here refers to the one who calls the check function. Additionally, the contract needs the outputs from all computation parties in order to generate the final result. Therefore, if there is

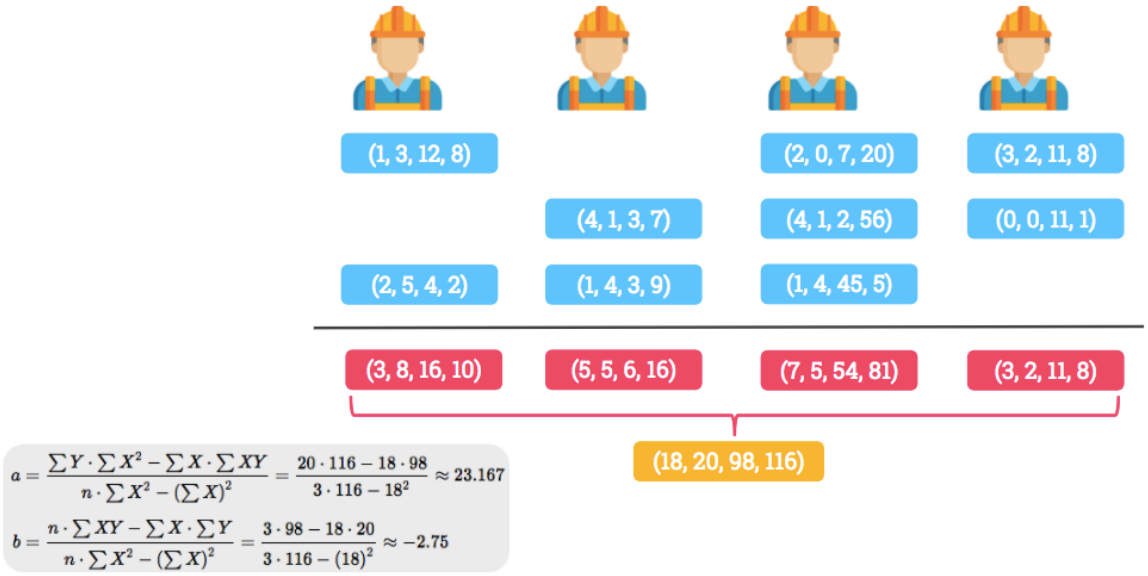
any computation party who has not submitted his output, the contract will not let the requester to check the result.

## G. Computation Party

### G.1 Compute

With reference to Section E.1's example, we have 3 raw data – (6, 5), (8, 2), (4, 13). After being processed by the data storers, they are split into a total of 9 shares.

| | Raw | Processed | Share 0 | Share 1 | Share 2 |
|---|---|---|---|---|---|
| Data 0 | (6, 5) | (6, 5, 30, 36) | (1, 3, 12, 8) | (2, 0, 7, 20) | (3, 2, 11, 8) |
| Data 1 | (8, 2) | (8, 2, 16, 64) | (4, 1, 3, 7) | (4, 1, 2, 56) | (0, 0, 11, 1) |
| Data 2 | (4, 13) | (4, 13, 52, 16) | (2, 5, 4, 2) | (1, 4, 3, 9) | (1, 4, 45, 5) |




$$a = \frac{\sum Y \cdot \sum X^2 - \sum X \cdot \sum XY}{n \cdot \sum X^2 - (\sum X)^2} = \frac{20 \cdot 116 - 18 \cdot 98}{3 \cdot 116 - 18^2} \approx 23.167$$

$$b = \frac{n \cdot \sum XY - \sum X \cdot \sum Y}{n \cdot \sum X^2 - (\sum X)^2} = \frac{3 \cdot 98 - 18 \cdot 20}{3 \cdot 116 - (18)^2} \approx -2.75$$

The above graph illustrates a possible situation after the data storers store the 9 shares. The 1st computation party receives 2 shares, the 2nd one receives 2 shares, the 3rd one receives 3 shares whilst the 4th one receives 2 shares. Here we ignore the date factor, but in the contract, each share will also associate with a date which is specified by the data storer when he stores the share (refer to Section E.1)

One thing worth mentioning here is that you can see no computation party holds adjacent shares of the same data, e.g. the 1st computation party holds (1, 3, 12, 8), therefore he does not hold (2, 0, 7, 20). In reality, it is possible that (3, 2, 11, 8) is distributed to the 1st computation party, instead of the 4th computation party (the situation in the graph). But for easier discussion, here we assume that the shares of the same data are distributed nicely to different computation parties.

To obtain $\alpha$ and $\beta$ of the linear regression analysis, each computation party first needs to summate the shares he has received. In the graph, there are 4 computation parties, therefore there will be 4 results – (3, 8, 16, 10), (5, 5, 6, 16), (7, 5, 54, 81), (3, 2, 11, 8). Then, the contract can further summate these 4 results, and get (18, 20, 98, 116). These values represent the summation of x, the summation of y, the summation

of xy and the summation of x square. Recall that the contract keeps track of the number of data stored, we can now apply the mathematical formula to obtain $\alpha$ and $\beta$.

$$\alpha = \frac{\sum x \sum y - \sum x \sum xy}{number\ of\ data\ \sum x^2 - (\sum x)^2} \qquad \beta = \frac{number\ of\ data\ \sum xy - \sum x \sum y}{number\ of\ data \sum x^2 - (\sum x)^2}$$

During the whole computation, the processed data – (6, 5, 30, 36), (8, 2, 16, 64) and (4, 13, 52, 16) are not exposed and hence we are able to privately preserve the raw data – (6, 5), (8, 2) and (4, 13).

As aforementioned, in the real situation, the contract takes the date factor into account. A requester may only want the data from 2017-01-01 to 2017-12-31. This can be done easily which the computation party will check whether the share's associated date is fall within the required range. If so, summate this share, otherwise ignore it. Technically speaking, the compute function contains a for-loop that performs this filtering.

Can we combine the role of requester and the role of computation party?

Every operation on the Ethereum blockchain comes with a cost – the gas. When the computation party calls the compute function, they have to pay Ether (calculated by the consumed gases times the gas price at that moment). Some may ask why we need the role of requester in this case? The requester can be a computation party and call the compute function by himself, rather than paying a request fee for others to call the compute function.

Let's see what will happen if we combine the 2 roles. Suppose there are several people who have the same goal – they want to conduct a linear regression analysis using the data from 1970 to 2000. Therefore, they register to be the computation parties together, call the compute functions, pay the costs, combine their outputs and obtain the final result.

Now there is a new person who would like to join the crew, but he has a different goal – he wants to conduct the analysis using the data from 2000 to 2010. The existing crew actually do not have any incentives to perform the computation again. They have already obtained the result they want and may not be interested in the period from 2000 to 2010. In fact, it is likely that they will withdraw from being a computation party after they have finished their computation.

Consequently, this new person now has to find others who have the same goal and form a new crew. Yet, under this circumstance, the data storers have to store their data once again for this new computation. It is highly inefficient if the data storers need to store the data again for every new computation. In a better design, the data storers may not have to store the data again, but the contract has to re-allocate the shares from the old crew to the new crew, which is still inefficient.

Thus, the reason why the requester cannot be the one who calls the compute function is because it is believed that they value the computation result more than the reward generated from computation. In other words, despite being a computation party can earn rewards, they do not have much incentive to stay after obtaining the result they want. Therefore, it is paramount to separate the 2 roles.

### G.2 Register

Every Ethereum node can register to be a computation party if he sends sufficient Ether to the contract, known as a deposit. The register function verifies two things.

1) whether the deposit is greater than or equal to (>=) the minimum amount of deposit
2) whether the caller is not in the list of computation party or in the pending list

If the above conditions are satisfied, the node will be put into the list of computation party or the pending list, depending on the situation. As mentioned in Section D, if the list of computation party is full, we will still collect the deposit but put the node in the pending list.

### G.3 Contribute

The contribute function can be called by the Ethereum node who has done the registration, regardless he is in the list of computation party or in the pending list. If a node wants to increase his deposit, he can call this function along with Ether, then the contract will add the amount to his deposit.

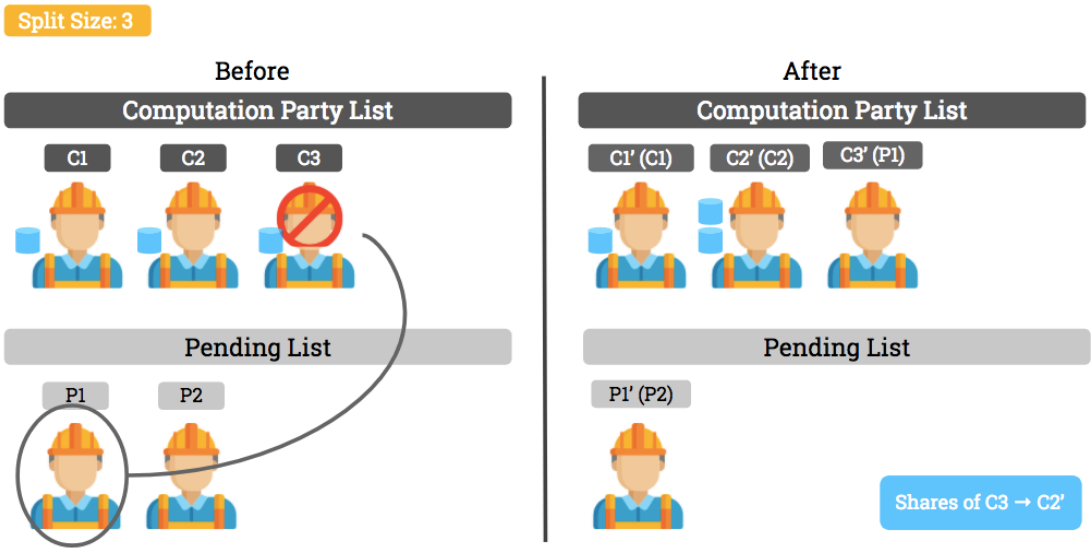
### G.4 Withdraw

The withdraw function can be called by the Ethereum node who has done the registration, regardless he is in the list of computation party or in the pending list. If he is in the pending list, the contract will simply send back his deposit, and remove him from the pending list.

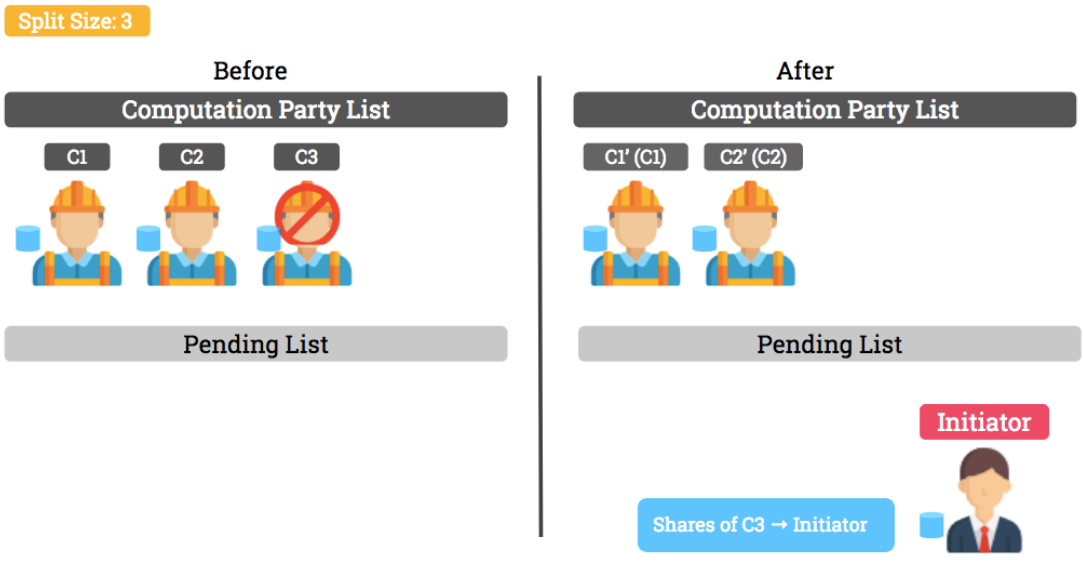
If he is in the list of computation party, the contract will send back his deposit while the list update's logic is as follows. The contract will remove the withdrawn node. Then, if there are nodes in the pending list, it will dequeue the 1st node in the pending list, and enqueue this node into the list of computation party.

Besides, the contract has to perform share re-allocation. Recall from Section E.1, the split size states how many shares that a data storer should split from a piece of data. For example, if the split size is 3, then the data storer has to split the data into 3 shares and store exactly 3 shares. After the list update, if the number of computation parties is still greater than or equal to (>=) the split size, then the shares held by this withdrawn party will be randomly re-allocated to the remaining computation parties. Otherwise, they will be re-allocated to the initiator.

Let's look at the example below. We assume that the computation party list can only have 3 nodes at maximum. Suppose C3 withdraws and the split size is 3. Since the pending list is not empty, we dequeue P1 from the pending list and enqueue him into the list of computation party. The new list of computation party has 3 nodes which is >= the split size, therefore the share held by C3 can be randomly distributed to either C1', C2' or C3', who are the old C1, C2 and P1. In the graph, it goes to C2'.




Another example is when the pending list is empty. The new list has only 2 nodes which is < the split size, therefore the share held by C3 goes to the initiator.




In the worst case scenario, all parties withdraw. Then, all shares will be re-allocated to the initiator. Even if this happens, it is still secure in a sense that the initiator cannot tell which share is split from which share.

# H. Initiator

### H.1 Contract Instantiation

In order to create a computation instance, the initiator has to provide 7 arguments. These are the rules that govern the data storer, the requester and the computation party. From Section D, E, F and G, they have been briefly mentioned and here is the summary.

| Arguments | Description |
|---|---|
| Deposit Threshold | The minimum amount of deposit that an Ethereum node needs to submit to the contract in order to register as a computation party |
| Computation Nodes Ceiling | The maximum no. of computation party. As mentioned in Section D, when the list of computation is full, for any subsequent registrations, we will put the nodes into the pending list. |
| Split Size | The no. of shares that a data storer must split from a piece of data. He must store exactly this amount of shares for a piece of data. |
| Computation Fees Threshold | The minimum amount of fee that a requester has to submit to the contract for a request. |
| Computation Period In Second | The duration of the computation period. |
| Checking Period in Second | The duration of the checking period. |
| Request Ceiling Per Requester | The maximum no. of request that a requester can submit to the contract within a round. |

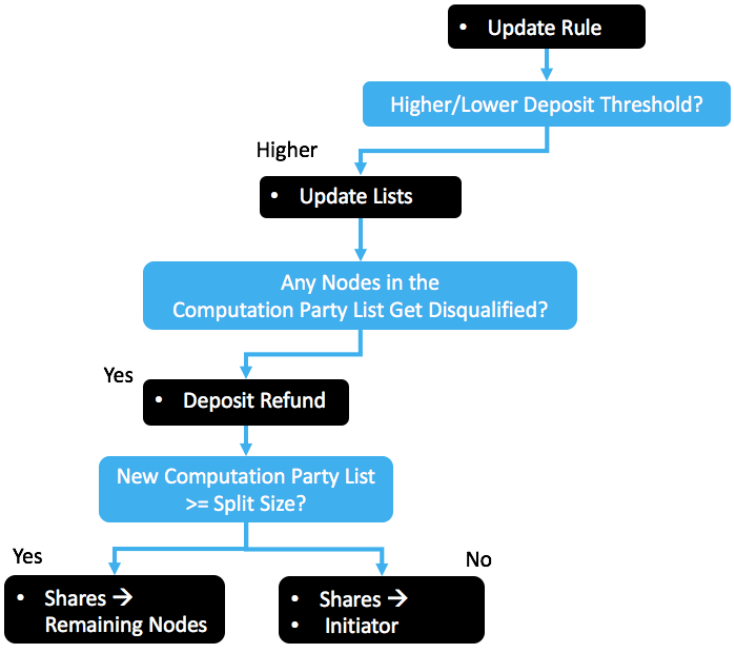
### H.2 Reset Rules

Over the time, the initiator may want to re-set some of the rules. The setter functions can be classified into two types. The first type will not result in any changes of the computation party list. They are the reset of 1) split size, 2) computation fees, 3) computation period in second, 4) checking period in second and 5) request ceiling per requester. When the initiator calls this type of setters, it will simply reset the corresponding rule. For example, if we reset the computation fees from $10 to $15, then any subsequent request submission must send with a fee of at least $15. If we reset the request ceiling per requester from 3 to 2, yet a requester has already submitted 3 requests before the reset, in this case, we will not remove 1 of his requests. However, for others who have not submitted 3 requests yet, they can only submit a maximum of 2 requests now.

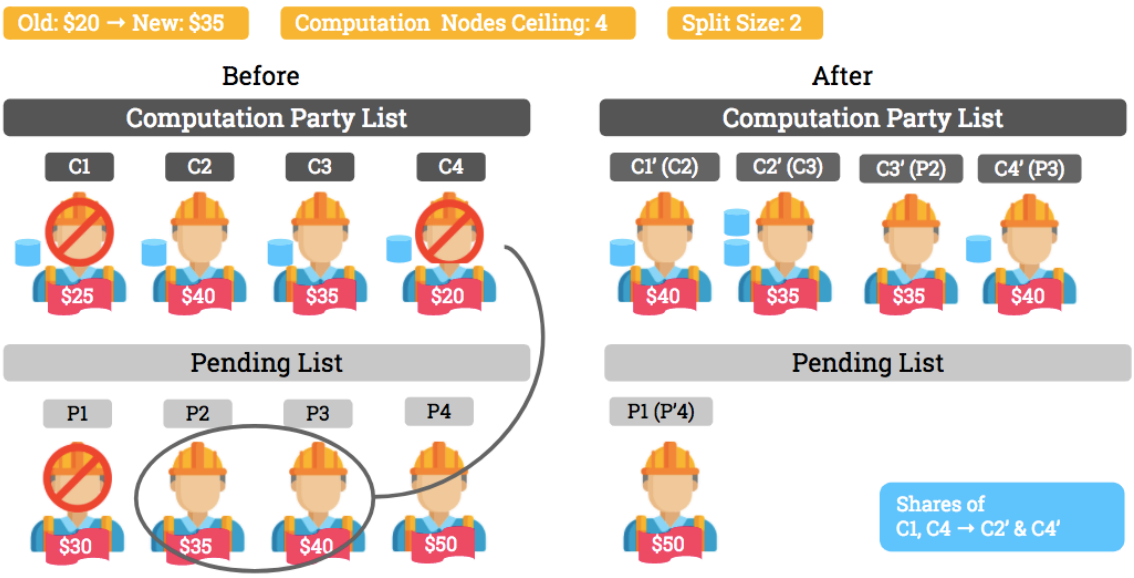
The second type of setters may make changes to the computation party list. They are the reset of 1) deposit threshold and 2) computation nodes ceiling. Given that the reset may bring changes to the list of computation party, the contract may perform share re-allocation.

Reset of Deposit Threshold

For the case of setting a lower deposit threshold, no list update needs to be done. Yet, if we set a higher deposit threshold, then the contract will first remove the nodes that do not meet the new requirement in both the list of computation party and the pending list, then send back the deposits to them. After the disqualification, if the pending list is not empty, we will enqueue the nodes in it to the list of computation party until the pending list is empty or the list of computation party is full. Finally, the contract will check if the new list of computation party has a size >= split size. If so, the shares held by the disqualified nodes will be re-allocated to the remaining nodes in the new list, otherwise to the initiator. The re-allocation logic is same as the one when a node withdraws. The graph below summaries the logic.



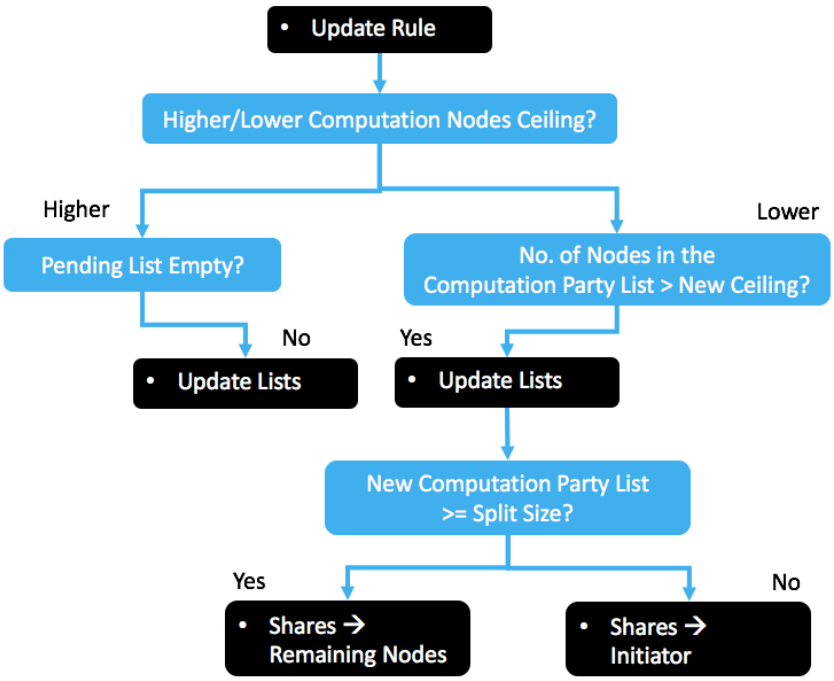

For example, suppose we re-set the deposit threshold from $20 to $35. In the graph below, C1, C4 and P1 will get disqualified. Since the pending list is not empty (which we have P2, P3 & P4), we enqueue them until the pending list is empty or the list of computation party is full. As the computation nodes ceiling is set to 4, here a full list of computation party means it contains 4 nodes. In this case, we can only enqueue P2 and P3, then the list of computation party will be full. Given that the split size is 2, and the new list has a size of 4, so the contract can randomly re-allocate the shares held by the disqualified nodes (C1 and C4) to the new C1', C2' C3' and C4' who are the old C2, C3, P2 and P3. In the graph, the 2 shares go to C2' and C4'.


Old: $20 → New: $35
Computation Nodes Ceiling: 4
Split Size: 2
Before
Computation Party List
C1
C2
C3
C4
$25
$40
$35
$20
Pending List
P1
P2
P3
P4
$30
$35
$40
$50
After
Computation Party List
C1' (C2)
C2' (C3)
C3' (P2)
C4' (P3)
$40
$35
$35
$40
Pending List
P1 (P'4)
$50
Shares of
C1, C4 → C2' & C4'

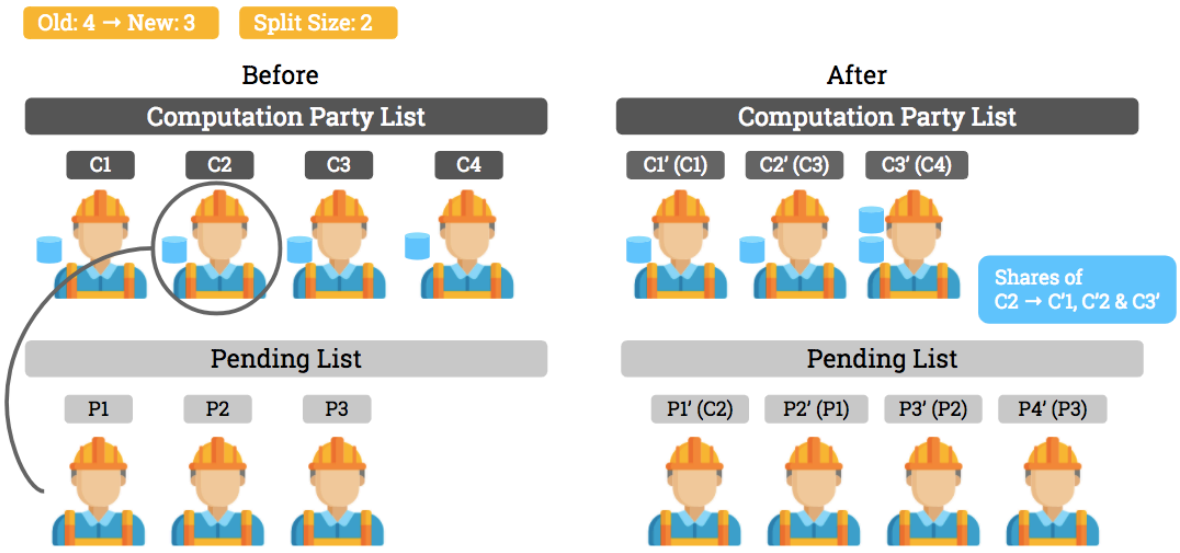
Reset of Computation Nodes Ceiling

For the case of setting a higher computation nodes ceiling, it means we can allow more computation parties. The contract will dequeue the nodes in the pending list and enqueue them to the computation party list until the pending list is empty or the computation party list is full. No share re-allocation is needed in this case.

But, if we set a lower computation nodes ceiling, and this new ceiling exceeds the current number of nodes in the computation party list. We will then randomly remove some nodes until the list of computation party has a size = the new ceiling. Here the removal will not lead to deposit refunds because we will insert these removed nodes at the head of the pending list. Thus, it is not a "disqualification" described in the deposit threshold reset. As they are in the pending list, we will still keep their deposits. Then, if the new list of computation party has a size >= split size, the shares will be re-allocated to the remaining nodes, otherwise to the initiator. The graph below summaries the logic.




For example, suppose we re-set the computation nodes ceiling from 4 to 3. In the graph, there are 4 nodes (C1, C2, C3 and C4) in the list of computation party, hence we will randomly pick one to remove it and then insert it at the head of the pending list. In this case, C2 is selected. Given that the split size is 2 and the new list has a size of 3, so the contract can randomly re-allocate the shares held by the removed node (C2) to C1', C2' and C3' who are the old C1, C3 and C4. In this graph, the share goes to C3'.


Old: 4 → New: 3
Split Size: 2
Before
Computation Party List
C1
C2
C3
C4
Pending List
P1
P2
P3
After
Computation Party List
C1' (C1)
C2' (C3)
C3' (C4)
Shares of
C2 → C'1, C'2 & C3'
Pending List
P1' (C2)
P2' (P1)
P3' (P2)
P4' (P3)

**H.3 Recognize Data Storer & Requester**

The initiator can call the addDataStorer and addRequester functions along with a person's address as an argument so as to recognize him to be a data storer and a requester respectively. Only after that, this person can call the data storer's functions or the requester's functions. In other words, if an unrecognized Ethereum attempts to call the functions of these roles, the execution will be rejected.

**H.4 Declare the Start of Computation Period & the End of Checking Period**

Only after the initiator calls the declareComputationStart function, the computation parties can call the compute function. If they attempt to call the compute function before it, the execution will be rejected.

The end of the computation period is determined by the moment when the initiator calls the declareComputationStart function, plus the computation period in second, which is one of the arguments when he instantiates the contract. During the computation period, no other operations are allowed. For instance, the data storer cannot store shares while the requester cannot submit requests.

The unofficial end of checking period is determined by the moment when the computation period ends, plus the checking period in second, which is one of the arguments when the initiator instantiates the contract. During the checking period, computation and other operations are disallowed. It is the buffering time for the requester to check the result.

The official end of checking period is the moment when the initiator calls the declareCheckingEnd function after the unofficial end of checking period. If he attempts to call this function before the unofficial end of checking period, the execution will be rejected. At the official end of checking period, all requests and results will be cleared. For computation parties who have called the compute functions properly, we label them as the "honest parties" and they will be rewarded. For those who have not, we label them as the "dishonest parties" and they will be penalized. The reward and and the penalty for each request is determined as follows.

$$reward\ per\ computation\ party = \frac{request\ fee}{no.\ of\ computation\ paty}$$

$$penalty\ per\ computation\ paty\ = \frac{reward\ per\ computation\ party * no.of\ honest\ party}{no.of\ dishonest\ party}$$

It can be seen that the more honest parties there are, the higher the penalty is. The reward will be sent directly to each honest party instead of adding it to their deposits. The penalty will be deducted from the dishonest party's deposit and then the total amount of penalty from all dishonest parties will be sent to the requester. If a dishonest party's deposit is below the penalty, the deposit will be the penalty in this case.

After deducting the penalty from a dishonest party, if his deposit is below the deposit threshold, we will disqualify him from being a computation party. The list update and share re-allocation logics are same as before. If the pending list is not empty, we will dequeue the nodes in it and enqueue them to the computation party list until the pending list is empty or the computation party list is full. After that, if the new computation party list >= the split size, the shares held by the disqualified nodes will go to the remaining nodes, otherwise to the initiator.

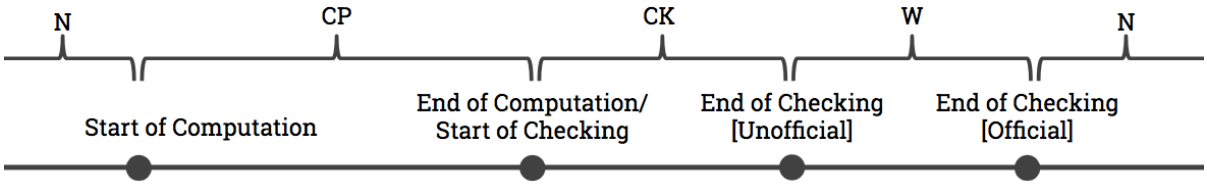
**H.5 Restore Shares**

Recall that the initiator will hold shares because of nodes' withdrawal, rules reset, etc. that leads to the list of computation party < split size. The restore function enables him to re-allocate the shares back to the computation parties when the list computation party >= split size. If he attempts to call this function when the list of computation < split size, the execution will be rejected.

One thing worth noting here is that whenever the initiator holds share, if he calls the declareComputationStart function, the execution will be rejected. In other words, the computation period cannot be started if the initiator holds shares.

## I. Summary of Functions

The following lists the functions of each role and when they can be called. CP stands for the computation period, CK stands for the checking period, W stands for the time after the unofficial end of checking period that the contract is waiting for the initiator to call the declareCheckingEnd function to officially end it, and finally N stands for the normal period.



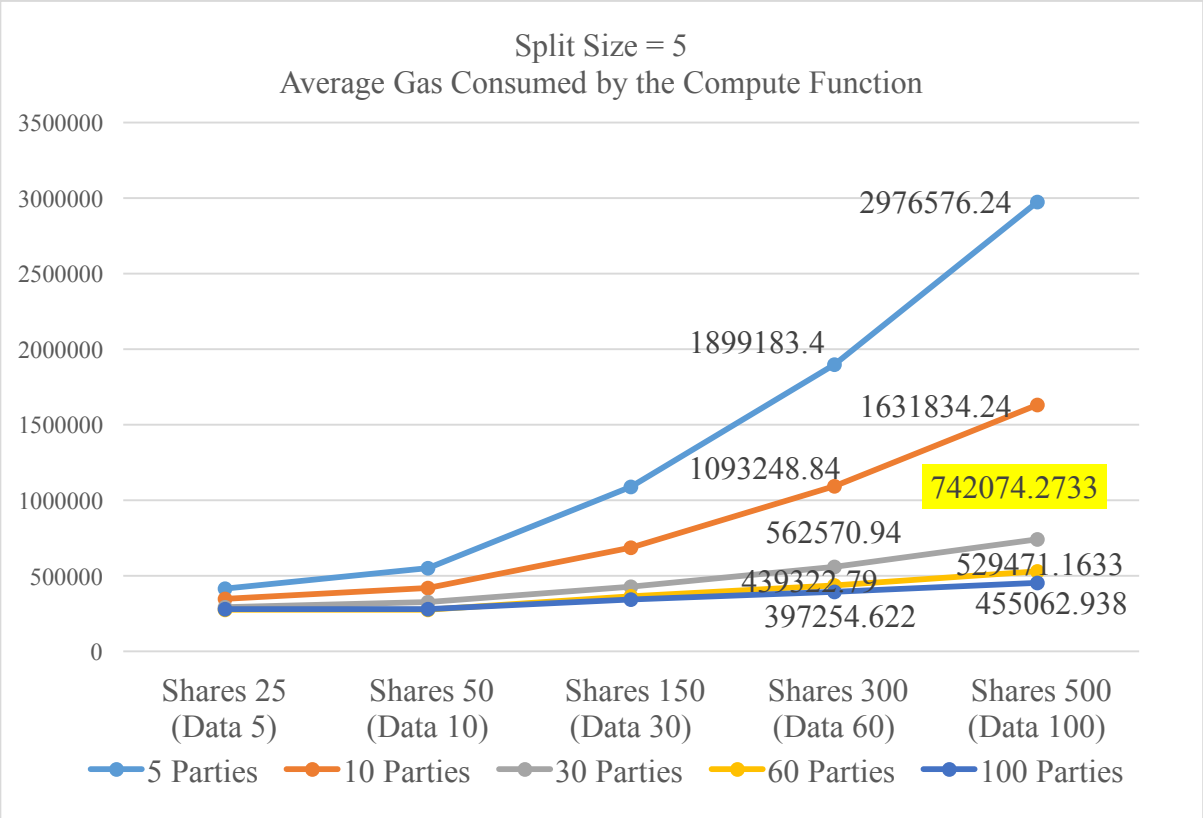

| Role | Responsibility | Function's Name | Period | | | |
|---|---|---|---|---|---|---|
| | | | CP | CK | W | N |
| Data Storer | 1. Store Shares | • store | ✗ | ✗ | ✗ | ✓ |
| Requester | 1. Submit Request | • submit | ✗ | ✗ | ✗ | ✓ |
| | 2. Cancel Request | • cancel | ✗ | ✗ | ✗ | ✓ |
| | 3. Check Result | • checkResult | ✓ | ✓ | ✓ | ✗ |
| Computation Party | 1. Compute | • compute | ✓ | ✗ | ✗ | ✗ |
| | 2. Register | • register | ✗ | ✗ | ✗ | ✓ |
| | 3. Contribute | • contribute | ✗ | ✗ | ✗ | ✓ |
| | 4. Withdraw | • withdraw | ✗ | ✗ | ✗ | ✓ |
| Initiator | 1. Instantiation | / | | | | |
| | 2. Reset Rules | • setDepositThreshold | ✗ | ✗ | ✗ | ✓ |
| | | • setComputationNodesCeiling | ✗ | ✗ | ✗ | ✓ |
| | | • setSplitSize | ✗ | ✗ | ✗ | ✓ |
| | | • setComputationFeesThreshold | ✗ | ✗ | ✗ | ✓ |
| | | • setComputationPeriodInSecond | ✗ | ✗ | ✗ | ✓ |
| | | • setCheckingPeriodInSecond | ✗ | ✗ | ✗ | ✓ |
| | | • setRequestCeilingPerRequester | ✗ | ✗ | ✗ | ✓ |
| | 3. Recognize Data Storer & Requester | • addDataStorer | ✗ | ✗ | ✗ | ✓ |
| | | • addRequester | ✗ | ✗ | ✗ | ✓ |
| | 4. Declare Start & End | • declareComputationStart | ✗ | ✗ | ✗ | ✓ |
| | | • declareCheckingEnd | ✗ | ✗ | ✓ | ✗ |
| | 5. Restore Shares | • restore | ✗ | ✗ | ✗ | ✓ |

# J. Performance

## J.1 Scalability

The graph below shows the average and the total gas consumed when the computation parties execute the compute function. Assume that a piece of data is split into 5 shares, when there are 100 pieces of data, the number of shares will be 500.

Suppose there are 30 computation parties. Then, each will consume 742074.2733 gases when he executes the compute function (refer to the yellow box below). As of Dec 2017, according to Ethereum Gas Station, the standard gas price is 4 Gwei (1 Gwei = 0.000000001 Ether). Hence, each will have to pay 0.00297 Ether, which is around $2.122 USD. The total gases consumed by all computation parties are 22262228.2, and hence the total cost of this computation will be 0.08905 Ether, which is around $63.67 USD.




## J.2 Security

The current splitting scheme is the additive secret sharing scheme. All shares are needed to reconstruct the data. The first n – 1 shares' values are set to be random positive integers. The last share is the value of the data minus the sum of all n – 1 shares' values. For example, if the data has a value of 5 and we want to split it into 3 shares. We will randomly pick 2 positive integers, say 4 and 9. Then the value of the last share will be 6 – 4 – 9 = -7.

The drawback of this scheme is that it requires the participation of all computation parties in order to calculate the final result which is fairly inefficient considering the gas price discussed in J.1. Therefore, a possible improvement of this project is to adopt the Shamir's secret scheme. Therefore, the data is split into n shares but we only require t+1 computation parties to reconstruct it back, where $t < n$.

## K. Conclusion

The blockchain technology provides a secure platform for MPC as it keeps track of all operations, e.g. data storage, computation, etc. More importantly, the change of state information brought by the operations cannot be maliciously reverted. Message exchange and identification of dishonest parties can hence be done in a more efficient manner.

The project realized some concepts described in Zyskind's MPC framework on Ethereum. Linear regression analysis is used as a conduit to explain the interaction between each role. Yet, the mechanisms described in this project such as when and how to punish dishonest parties is believed to be applicable to other types of computations.

There are 4 roles – the data storer, the requester, the computation party and the initiator, as well as 4 critical moments – the start of computation period, the start of checking period [/the end of computation period], the unofficial end of checking period and the official end of checking period. Each role has their own functions that can only be called if the time is right. Otherwise, the contract will reject the execution.

Lastly, improvements can be done by using the Shamir's secret scheme rather than the additive secret scheme, therefore results can be generated without the participation of all computation parties, consequently lowering the cost of computation.

## L. References

Zyskind, G. (2016). Efficient Secure Computation Enabled by Blockchain Technology. Massachusetts Institute of Technology. Retrieved from https://dspace.mit.edu/handle/1721.1/105933

ETH Gas Station. Retrieved from https://ethgasstation.info/

## M. Appendix

Given that generic types and many common utilities are not supported in Solidity yet, I have implemented my own libraries in this project. Also, considering each role has their own logic, therefore there are several contracts and each represent a part of the system. Here are some brief descriptions.

| Contract | Description |
|---|---|
| Registrable | • Contains logic of registration, withdrawal, etc. |
| RegressionDataStorage | • Inherits Registrable<br>• Contains logic of share re-allocation, data storage, etc. |
| Computable | • Contains logic of start & end declaration |
| DateBasedComputable | • Inherits Computable<br>• Contains logic of request submission, cancellation, etc. |
| RegressionDataComputation | • Inherits RegressionDataStorage & DateBasedComputable<br>• Contains logic of computation, rewarding, penalizing, etc. |

| Library | Description |
|---|---|
| AddressQueue | • Data structure used to maintain and manipulate the computation party list, data storer list & requester list |
| DateChecker | • Utility function used to verify date |
| RegressionShareManagement | • Data structure used to store and manipulate shares |
| DateBasedRequestManagement | • Data structure used to store and manipulate requests |
| RegressionResultManagement | • Data structure used to store and manipulate results |