A decentralized private data marketplace using blockchain and secure multi-party computation

Julen Bernabé-Rodríguez*[1], Albert Garreta[2], and Oscar Lage[1]

[1]TECNALIA, Basque Research and Technology Alliance (BRTA), Bizkaia Science and Technology Park, Building 700, E-48160 Derio, Bizkaia, Spain, *julen.bernabe@tecnalia.com*
[2]Basque Center of Applied Mathematics, Bilbao, Spain, *agarreta@bcamath.org*

March 20, 2022

Abstract

Big data has proven to be a very useful tool for companies and users, but the widespread cloud computing architecture has ended in third-parties storing and processing the owners' private data. Secure multi-party computation (SMPC) allows the data owners to jointly train arbitrary models on their private data while assuring privacy, and thus gives owners the data control back. Besides, with a blockchain it is possible to coordinate and audit those computations in a decentralized way.

A private data marketplace is a space where researchers and data owners meet to agree the use of private data for model trainings. This document presents a candidate architecure for a private data marketplace by combining SMPC and a public, general-purpose blockchain. Such a marketplace is proposed as a Smart Contract deployed in the blockchain, while the privacy preserving computation is held by SMPC.

Keywords— Multi-party computation, Blockchain, Security, Privacy, Edge computing, Distributed computation, Data economy.

1 Introduction

The impressive growth of data science has led to the creation of companies that store, process and manage data from smaller companies who cannot afford to do it by themselves, starting what today is called as cloud computing. Over the years, cloud computing has spread, making cloud companies more and more powerful. However, this architecture has two major drawbacks to consider. On the one hand, data owners do not have the control of their private data, and besides must pay to cloud companies so as to use their services. On the other hand, as all data items are stored in the same place, we have a single point of failure that attackers can exploit.

A natural solution to this problem is to decentralise the location of the private data. This approach, frequently called edge computing, involves devices which are able to collect, store, and process data locally, without the need of third-parties. Edge devices allow companies to process their private data simultaneously in different locations, and thus the processed results end up distributed. Gathering all of the results without a cloud party is not trivial at all, though. Secure multi-party computation (SMPC) is key at this stage, as it lets the data owners evaluate an algorithm on their data while guaranteeing confidentiality on the inputs used. Moreover, with this technology, companies can process their private data along with private data from other companies.

Blockchain is a brand new technology which provides decentralization, integrity, non-repudiation and traceability of computations. Blockchains have native tokens that can be used in different ways inside them. Combining this technology with SMPC, we can build a marketplace upon a blockchain, where parties can meet each other and enroll in model trainings for an economic incentive using their private data. This opens new paradigms regarding private data economy.

For the sake of simplicity, we consider a marketplace where there are two user profiles: *data scientists* and *data owners*. Data scientists have a model they want to train, while data owners have private information that could be useful for the data scientists' models. The scientists want to meet data owners who can improve their model, and the latter want to take some profit from their private data, while assuring that its information will never be leaked. We will denote each model training process a *model campaign* or simply a *campaign*[1] from now on.

We only consider Out of Service (OoS) errors in the marketplace. This assumption is logical from the fact that data owners are risking their private inputs for an economic incentive. Therefore, we can assume that their goal is not to learn from the inputs of the other data owners, but just to receive some reward for sharing their data. This implies that only one player might be interested in knowing the private inputs of the others: the data scientist.

The main contribution of this paper is to propose a candidate for a decentralized private data marketplace combining SMPC and blockchain technology. For this, we give a detailed explanation of the SMPC protocol implemented in SCALE-MAMBA [8] and suggest a grouping protocol based on the Bin Covering Problem [1] that reduces OoS errors in the marketplace. Moreover, we avoid the need of third parties, which results in a marketplace where all users are naturally incentivized to participate. This was unsolved in previous work.

More precisely, Section 2 introduces SMPC and blockchain, and studies how to merge them in order to construct the marketplace and analyzes the prior work related to this field. Section 3 explains with all due detail how the marketplace and the grouping/SMPC protocols work. Along that section, we consider the incentives players have to enroll in campaigns and work out the rewarding process too. On Section 4 we give some conclusions and further work.

In this work, all users participating in the network are assumed to be running at least an edge device. This marketplace must not be understood as a fully open protocol: only users with enough computation power will be able to deal with difficult SMPC models. To relax this condition, we introduced reputations. Only players with enough reputation will be able to participate in difficult campaigns. We will see how this works in Section 3.

2 Technical background

2.1 SMPC and SCALE-MAMBA

Secure multi-party computation allows for a set $P = \{P_1, \ldots, P_n\}$ of n players to perform an arbitrary computation using the private inputs of the players, even if some of them are corrupted. This technology lets the players learn the output, while no information but the one derived from such output is leaked. In consequence, the data scientist and data owners can use SMPC for training the data scientist's model, while preserving the data owners' inputs private.

In the literature, it is typical to define the adversary as an entity able to corrupt some of the players. Adversaries able to make players deviate from the SMPC protocol are called *malicious* adversaries, whereas those who can only learn from their inputs without making them deviate from the protocol are called *semi-honest*. An SMPC network is said to be a *honest-majority* network if the adversary cannot corrupt more than a half of the players. Otherwise, it is said to be a *dishonest-majority* SMPC network.

In this paper, we consider SMPC protocols for a honest-majority network with malicious adversaries, as they fit perfectly with our situation and turn to be more efficient. These protocols are usually based on Linear Secret Sharing Schemes (LSSS), so we will only consider LSSS-based SMPC protocols in this document.

Currently, there are several SMPC implementation projects in course [6]. In the present work we use SCALE-MAMBA due to its long history in the field and due to the advanced state of development of its software. SCALE-MAMBA supports four instances of SMPC protocols:

- Full-Threshold.
- Shamir sharing.
- Replicated sharing.

[1] Section 2.2.1 introduces an intuitive description of what a campaign is.

- General $\mathsf{Q}_2 - MSP$ sharing.

Among the above, we use the Shamir sharing setting because it is a quite simple protocol which is secure against a honest-majority network with malicious adversaries. We will refer to it as the Shamir Protocol. The main documentation and a summary of this can be found in [8].

Since the Shamir Protocol works for a honest-majority network, it succeeds if and only if at least $n/2$ players behave correctly. If this condition is not fulfilled, the execution ends with an **ABORT** flag. In SMPC literature, these conditions are called *threshold conditions*. Moreover, if a player gets out of service during the SMPC execution, all players will abort too. This is why our main concern throughout this paper will be to reduce OoS errors inside the marketplace.

In the Shamir Protocol, the model is given as a circuit that players have to evaluate. Intuitively, this protocol uses LSSS to split players' private data into so-called *shares*, and then:

1. The shares are distributed among the players.
2. Each player performs the computations on the shares received from Step 1 following the circuit.
3. The final shares from Step 2 are combined to reconstruct the output.

The Shamir Protocol divides the computations into two *phases*. In the *offline phase*, the players do all the expensive interactions between each other, in order to be prepared for the *online phase*. The online phase is where the circuit evaluation happens. In other words, the steps above occur in the online phase, while other heavy and previous computations are done before.

2.2 Blockchain

As mentioned in Section 2.1, SMPC and in particular SCALE-MAMBA are useful tools for training a model on private data. SCALE-MAMBA, in fact, gives us the ability to train models that involve private datasets, while preserving their privacy.

Nevertheless, SCALE-MAMBA has a major drawback: players are considered to know each other before the computations. This means that scientists who want to train their models must know the data owners before setting up the network. Moreover, once the network is set up, they cannot add or remove players even if they want.

As a result, we need to integrate SCALE-MAMBA with another technology that serves us as a place for connecting players. This platform will take care of orchestrating the model campaigns, and automatize their whole process. What is more, the platform is intended to manage payments intrinsically. This is why we shall call this platform *the marketplace* from now on.

More precisely, we want this marketplace to be:

- Open: anyone can access it.
- Borderless: users can access it wherever they are.
- Neutral: the key is the dataset value of the data owners, regardless of who they are.
- Censorship-resistant: there is no possibility of banning people in the marketplace.
- Public: everyone in the marketplace can verify that someone participated in previous campaigns. This way, all players will have a reputation that can be key for future model trainings.
- Payment-suited: we want payments to be natural.
- Turing-complete: the marketplace must be able to orchestrate the whole campaign process.

Public blockchains are decentralized and distributed digital ledgers used to record transactions among many computers. These transactions are grouped inside *blocks*, which when validated are added to the ledger, forming a *chain*. This way, blocks that are validated and added to the chain cannot be changed. The above features make public blockchains open, borderless, neutral, censorship-resistant, public and payment-suited.

The Ethereum blockchain is a good choice for setting up such a marketplace. Ethereum stores both data and code, and is able to run that code and store the results of the execution. It is, in fact, turing-complete. From a practical point of view, this means Ethereum is a decentralized computer that executes programs. Each execution must be paid in the local currency, called *ether*. The programs are called *Smart Contracts*,

and enable users to develop decentralized applications with intrinsic economic properties. Ethereum allows us to create a Smart Contract deploying our marketplace, thus transfering to it all the properties mentioned before.

2.2.1 Design of the marketplace

Let us merge the above technologies to design a private data marketplace. As stated before, the main purpose of the marketplace is to orchestrate the different model campaigns happening inside. We intuitively consider the following cycle for a campaign in the marketplace:

1. A data scientist starts a campaign by registering an SMPC request on the blockchain.
2. All data owners receive the campaign and decide if they sign up or not. Those that want to participate sign up for the campaign on the blockchain.
3. After some time, the scientist ends the subscribing period and decides which data owners will participate in the SMPC execution.
4. The selected players perform the SMPC.
5. Data owners that performed the SMPC are paid for their services.
6. The process is repeated again.

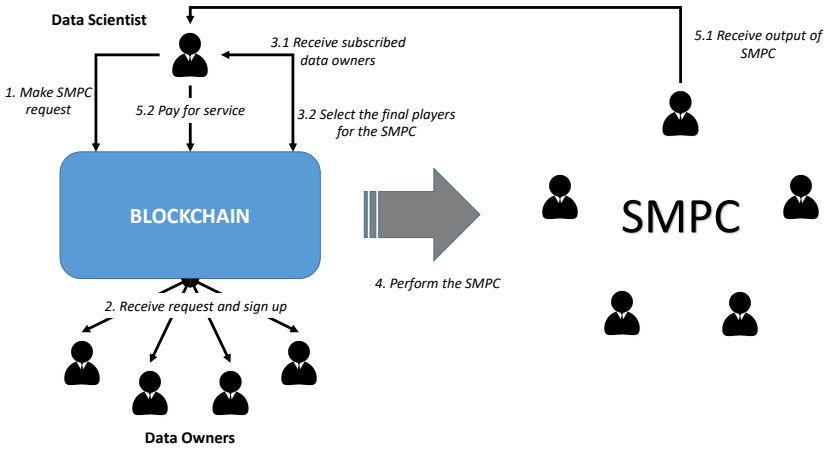

Figure 1: The cycle of a campaign in the marketplace.

Thus, the marketplace can be understood as a large *to-do-list*. In to-do-lists, we write down tasks that remain to be done, and mark them as done when we finish them. The marketplace works in a similar way. When a data scientist starts a new campaign, this is added to the end of the to-do-list managed by the marketplace. Once the data owners are subscribed, the SMPC is executed and the campaign is marked as finished in the to-do-list. With this design, the marketplace is able to manage several campaigns simultaneously. Moreover, finished campaigns can be reused.

We designed and implemented a Smart Contract deploying the private data marketplace. The Smart Contract implemented is written in Solidity language, and imports another contract implementing a token written by the OpenZeppelin Community [16]. In order to interact with it, we implemented a client written in Python that contains the SCALE-MAMBA SMPC framework. The client allows users to abstract from the interaction with the marketplace and from the SMPC execution. The client can be deployed as a Docker container, and integrates other tools such as IPFS [17], Eventeum [18]... For a detailed study of this client and the Smart Contract, refer to the GitHub repository in [19].

2.3 Related work

The main contribution of the present work with respect to other related projects consists in:

- A higher level of decentralization and openness attained.
- The design of a system of incentives between algorithm owners and data owners.

Related work and projects can be found in [9, 10, 11]. In general, these projects rely on separate computation nodes taking over the SMPC execution, something that does not occur in our design, where there are only two types of market participants: data owners and data scientists. The computing nodes are assumed to satisfy certain requirements, with such requirements resulting in a focus on scalability at the expense of decentralization and openness. An example of the this is the fog computing project White-City [10]. Also in KRAKEN [9] the SMPC computation is excuted by dedicated nodes. Here all participants are authenticated, and thus know each other, hence opting to significantly favour scalability in exchange for decentralization. The ARPA project [11] is another fog computing approach for the marketplace. In this case computation nodes are elected pseudo-randomly, and will not always be the same. Consequently, this is arguably a more decentralized approach for the marketplace. However we point out that the current implementation involves a coordinator that has some key functions inside the network. Thus, the network becomes centralized around that figure.

On a different note, none of the projects mentioned here have clearly discussed incentive mechanisms, something that we address in the present work. Having only two types of market participants, instead of three as in the projects above, designing and analysing incentive mechanisms becomes more feasible.

3 The cycle of a campaign

This section will deepen inside the steps described in Section 2.2.1. We will give exact descriptions of the conditions needed to make the whole cycle of a campaign secure.

As said before, the marketplace will give data owners and data scientists some reputation. By doing this, a user can consider whether the reputation another user has is enough or not, and then decide if it is worth engaging in the campaign with them. Thus, we can prevent users with limited computing resources from participating in difficult computations.

3.1 Starting a campaign

In the campaign cycle, the first to move is the data scientist by registering an SMPC request. The request will be registered inside a campaign, which will be published in the marketplace[2]. For the SMPC execution to be as secure as posible, the data scientist will have to establish some conditions to the campaign. First of all, when training a model, the size of the dataset is of vital importance for the data scientist. A mean among 9 salaries has not the same value than a mean among 10000 salaries. Considering this fact, the scientist must establish the *Minimum Overall Dataset Length* (MODL) they require for the model they want to train.

In addition, the scientist must indicate the amount of money they will pay if computations finish successfully. This information and the scientist's reputation are key for data owners to decide whether they sign up for the campaign or not. A well paid job is obviously more atractive for the data owners. Besides, more complex models (or models that require larger MODL's) should be better paid than simpler ones.

Finally, the data scientist would like to be assured that the data owners applying to their campaign can deal with its computing difficulty. In other words, if the data scientist wants to train a complex model, they would like to train the model with data owners that are known to perform the SMPC succesfully. This is measured with players' reputation. But recall that, at this stage, data owners are not subscribed yet. Thus, to prevent players with low reputations from signing up for the campaign, the scientist must establish a *minimum required reputation*. If a data owner does not reach this minimum reputation, they will not be able to subscribe.

Summarizing, when starting a new campaign, the data scientist has to establish the following parameters:

- The MODL.
- The total amount of tokens to be paid. We let this amount be $\mathcal{P}$. In Section 3.2 we will see how many tokens are paid to each data owner.

[2]In fact, the campaign is added to the end of the to-do-list managed by the Smart Contract.

- The minimum reputation required for a data owner to sign up. The data scientist will stake this reputation.

3.2 Signing up for a campaign

Once the campaign is published, data owners receive it and decide if they sign up or not for it. Suppose a data owner receives the campaign and has enough reputation to sign up for it. Let that campaign have MODL $\mathcal{L}$ and total payment $\mathcal{P}$. Then each data item will be paid, approximately, with $\mathcal{P}/\mathcal{L}$ tokens. If the size of the data owner's dataset is w, the data owner will receive $\hat{\rho}$ tokens, where $\hat{\rho} \in (\rho/2, \rho]$ and

$$\rho = \frac{\mathcal{P}}{\mathcal{L}} \cdot w. \tag{1}$$

Note that ρ is known at the subscribing time, so a data owner can decide whether to sign up for a campaign based on it. In other words, ρ gives a hint about the amount of tokens the data owner will receive for their services. The proof for $\hat{\rho} \in (\rho/2, \rho]$ is shown in Section 3.2.1. Thus, since ρ is known at this stage, data owners can decide if this campaign is well paid before subscribing to it.

In order to assure that $\hat{\rho} \in (\rho/2, \rho]$, the size of each data owner's dataset is required to be, at most, $w < \mathcal{L}$. This condition is actually useful since it allows us to assure that there are at least two data owners signed up to perform the computations. When combined with the data scientist, we have at least three players to perform the SMPC. The Shamir Protocol inside SCALE-MAMBA needs at least three players to perform the SMPC correctly, so we kill two birds with one stone.

Finally, suppose data owners receive a campaign they think is well paid. Suppose they have enough reputation to subscribe to it but the scientist has not good reputation. If data owners do not consider this fact, the SMPC execution might not end succesfully. Hence, we can expect data owners to sign up only for well paid campaigns from scientists with sufficiently good reputation.

In conclusion, when signing up for a campaign, data owners must consider the following facts:

- They must reach the minimum required reputation established by the scientist. Data owners will stake that reputation[3].
- Their dataset must be of size $w < \mathcal{L}$.
- The campaign is well paid, according to the data owner's criterion.
- The scientist has good enough reputation, according to the data owner's criterion.

It is important to mention that not all subscribed data owners might be selected for the SMPC execution. We will see this in Section 3.3.

3.2.1 The actual payments

Let us see why data owners know they will receive between $\rho/2$ and ρ tokens in a campaign before signing up for it. We consider a data owner who wants to sign up for a campaign which has MODL $\mathcal{L}$ and total payment $\mathcal{P}$. We let the size of the data owner's dataset be w. Then $\rho = \frac{\mathcal{P}}{\mathcal{L}} w$ from the above section, and we denoted the real payment as $\hat{\rho}$.

The discrepancy between ρ and $\hat{\rho}$ is due to the fact that the actual overall dataset length may vary from the MODL established by the scientist. Indeed, suppose a scientist published a campaign with MODL $\mathcal{L}$ and that the data owners who signed up had dataset lengths $W = \{w_1, \ldots, w_n\}$, where $w \in W$. Then, the subscribing time does not finish until

$$\sum_{j=1}^{n} w_j \geq \mathcal{L}.$$

This means that there will likely be more than $\mathcal{L}$ data items when the subscribing time is over, and thus the exact final payment is not computed with the formula in (1), but with

$$\hat{\rho} = \frac{\mathcal{P}}{\sum_{j=1}^{n} w_j} w$$

where $\sum_{j=1}^{n} w_j \geq \mathcal{L}$.

[3]They can put in stake more reputation than the minimum established by the scientist.

Let us see why $\hat{\rho} \in (\rho/2, \rho]$. On the one hand, we can easily observe that the largest possible $\hat{\rho}$ is attained when $\sum_{j=1}^{n} w_j = \mathcal{L}$. In this situation we have:

$$\hat{\rho} = \frac{\mathcal{P}}{\sum_{j=1}^{n} w_j} w = \frac{\mathcal{P}}{\mathcal{L}} w = \rho.$$

On the other hand, recall from the above section that we required each w_j to be $w_j < \mathcal{L}$ for $j \in \{1, \ldots, n\}$, and that the sign up process finishes once there are enough players $P_1, \ldots, P_n$ such that

$$\sum_{j=1}^{n} w_j \geq \mathcal{L}.$$

Considering these conditions, it is easy to see that the worst case payment occurs when:

- The first $n-1$ players satisfy that
$$\sum_{j=1}^{n-1} w_j = \mathcal{L} - 1.$$
- The n-th player that signs up for the campaign has dataset length $w_n = \mathcal{L} - 1$.

Consequently, in this case we have that $\sum_{j=1}^{n} w_j = 2\mathcal{L} - 2$. In this situation, the final payment is:

$$\hat{\rho} = \frac{\mathcal{P}}{\sum_{j=1}^{n} w_j} w = \frac{\mathcal{P}}{2\mathcal{L} - 2} w > \frac{1}{2}\rho.$$

As a result, when signing up for the campaign, data owners know that they will receive $\hat{\rho}$ tokens, where:

$$\hat{\rho} \in \left(\frac{1}{2}\rho, \rho \right].$$

3.3 The grouping protocol

A campaign enters this stage when $\sum_{j=1}^{n} w_j \geq \mathcal{L}$. At this moment, the data scientist has to select the players that will participate in the SMPC execution. Recall from Section 2.1 that in SCALE-MAMBA, if a player suffers from an OoS error during the SMPC execution, all players will abort the protocol. To reduce the scope of this problem, we propose to split the players into subgroups. One player suffering from an OoS error would only output an abort in one subgroup, not affecting the other SMPC executions. This way, we assure that most of the subgroups end up succesfully, by isolating the aborts. This, combined with the fact that all subscribed data owners reach the minimum reputation required, reduces the chances of our SMPC executions failing. With this in mind, we give a grouping protocol that maximizes the number of subgroups to make.

We introduce the following notation for the present section. Let n be the number of data owners that signed up for the model campaign. We will denote them as $P_1, \ldots, P_n$. We also let w_i be the size of the database of P_i. These database lengths will also be called *weights* in this section. Finally, we will refer to the aforementioned subgroups as *bins* from now on.

For a data scientist to group all data owners in several bins, there are some conditions to keep in mind. Some of these conditions may sometimes be incompatible, so we need to find a balance for them to be fulfilled as precisely as possible.

- ***Condition 1:*** *Data owners with large databases must end up assigned to some bin.* Observe that a data owner with a large w_i is more interesting to the data scientist.
- ***Condition 2:*** *Data owners with bad reputation will not be admitted.* A data owner that has successfully participated in SMPC executions before is less likely to suffer from an OoS error.
- ***Condition 3:*** *Bins with a large number of players must be taken with care, or even discarded.* A bin with many players has more possibilities to suffer from OoS errors. This condition can be relaxed if Condition 2 is satisfied.
- ***Condition 4:*** *The data scientist must always be in each bin.* If not, they will not be able to obtain the outputs of all bins.

- ***Condition 5:*** *At least two data owners are required to form a bin.* Combining this with Condition 4, we assure three players per bin. This condition comes from the threshold condition of the Shamir Protocol: we need at least three players to perform an SMPC execution.
- ***Condition 6:*** *The number of bins must be as large as possible.* The more bins the players are splitted into, the greater probability of obtaining outputs for the scientist.
- ***Condition 7:*** *Each bin must have an acceptable number of data items.* A model trained with a tiny dataset is not insteresting at all.

Taking into account the above items, the grouping protocol we are searching for can be roughly formulated as a *Bin Covering Problem*. The latter problem is defined as follows: one is given a list of elements $L = \{e_1, \ldots, e_n\}$ where each element has size $s(e_i) \in \mathbb{Z}\ \forall i \in \{1, \ldots, n\}$. One is also given an integer $C \geq 1$. A *packing* of L into m bins is a partition of L into m subsets

$$L = B_1 \cup \cdots \cup B_m$$

such that

$$\sum_{e \in B_i} s(e) \geq C \text{ for all } i \in \{1, \ldots m\}.$$

The goal of the Bin Covering Problem is to find the maximum m for which a packing of L into m bins exists.

Let us redefine the problem by setting the elements e_i as players P_i and the sizes per player $s(P_i)$ as w_i. Then, when maximizing m we fulfill Condition 6. Condition 4 is straightforward. To satisfy Conditions 3 and 7, the capacity of the bins C must be specified. We will discuss this later in this section, and assume now an arbitrary C such that $1 \leq C < \mathcal{L}$.

Condition 2 is straightforward from Section 3.1: the scientist required a reputation threshold to participate in the campaign, and thus there are not players with bad reputation. Furthermore, when giving an algorithm for the Bin Covering Problem, we will see that Conditions 1 and 5 will be satisfied too.

The Bin Covering Problem was firstly studied by Assmann in [1], and Assmann *et al.* showed in [2] that the problem is *NP-hard*. They besides considered the first approximation algorithms, proving their worst-case performances too. Csirik *et al.* provided two new approximation algorithms for solving the problem in [3]. Those algorithms are quite simple and practical, and they give a good approximation of the optimal result. In [4] and [5] other algorithms improving the approximation algorithms in [3] were given, but those required a great number of items to make a difference. Thus, they are considered only theoretically interesting for our use case. In fact, we will use the *Improved Simple heuristic* described in [3] to group the players into bins.

In the protocol below, when assigning a player to a specific bin, we will say *place / add / introduce / remove an element* by abuse of terminology. By *filling* a bin, we understand adding elements until the capacity of the bin is equal or greater than C. Finally, when talking about the *weight of a bin* or *bin capacity*, we are referring to the number of data items assigned to that bin.

When introducing protocols in the paper, we will use boxes as the one below. These boxes represent classes in an object-oriented programming language. The grouping protocol, thus, is the following:

Protocol GROUPING

Let $P = \{P_1, \ldots, P_n\}$ be a set of players whose respective weights are $W = \{w_1, \ldots, w_n\}$. We aim to pack them into bins of common weight at least C.

1. Compute $\hat{w}_i = \frac{w_i}{C}$ and sort the $\hat{w}_i$ in a decreasing order, so that we have:

   $$\hat{w_1} \geq \hat{w_2} \geq \cdots \geq \hat{w_n}.$$

   Observe that the indexes have been redefined at this stage.
2. Those $\hat{w}_i \geq 1$ are placed each one in an empty bin. Analogously, the element with smallest weight is placed in one of those bins. This way, we satisfy Conditions 1 and 5.

3. At this stage, all $\hat{w}_i$ satisfy that $\hat{w}_i < 1$. By abuse of notation, we rename this ordered list as

$$1 > \hat{w_1} \geq \hat{w_2} \geq \cdots \geq \hat{w_{n'}}.$$

Split the above ordered list into three sublists so that:

$$\begin{cases} 1 > \hat{w}_1 \geq \hat{w}_2 \geq \cdots \geq \hat{w}_k \geq 1/2 \\ 1/2 > \hat{w}_{k+1} \geq \hat{w}_{k+2} \geq \cdots \geq \hat{w}_m \geq 1/3 \\ 1/3 > \hat{w}_{m+1} \geq \hat{w}_{m+2} \geq \cdots \geq \hat{w}_{n'} \end{cases}$$

By abuse of notation, we redefine and reorder these sublists as follows:

$$\begin{cases} 1 > \hat{x}_1 \geq \hat{x}_2 \geq \cdots \geq \hat{x}_{k_1} \geq 1/2 \text{ (called X sublist)} \\ 1/2 > \hat{y}_1 \geq \hat{y}_2 \geq \cdots \geq \hat{y}_{k_2} \geq 1/3 \text{ (called Y sublist)} \\ 1/3 > \hat{z}_1 \geq \hat{z}_2 \geq \cdots \geq \hat{z}_{k_3} \text{ (called Z sublist)} \end{cases}$$

where $k_1 + k_2 + k_3 = n'$. Hence we let $X = \{\hat{x}_1, \ldots, \hat{x}_{k_1}\}$, $Y = \{\hat{y}_1, \ldots, \hat{y}_{k_2}\}$ and $Z = \{\hat{z}_1, \ldots, \hat{z}_{k_3}\}$. The reordering we have done works as follows: in X and Y, if two weights have the same value, then the weight with greater reputation is considered greater. In Z, instead, in the same case, the weight with smaller reputation is considered greater. This reordering gives some election preference to those players who gambled more reputation than the minimum which was established by the scientist.

4. Do the following while $X \cup Y$ or Z are non-empty lists:
   (a) If $\hat{x}_1 \geq \hat{y}_1 + \hat{y}_2$, remove $\hat{x}_1$ from X and place it in an empty bin. Otherwise, remove $\hat{y}_1$ and $\hat{y}_2$ from Y and introduce them in an empty bin.
   (b) Fill the bin opened in the previous step by removing elements from the end of the sublist Z and adding them to the bin. If the elements in Z are not enough to fill the bin, take an element from Y to fill it.

   Note that Conditions 1 and 5 are satisfied for the bins filled at this step.
5. We have two cases:
   - If $X \cup Y = \emptyset$: take the first element in Z and the unfilled bin (if there is not such a bin, take a new bin), remove that element from Z and place it in the bin. Repeat this process until all elements in Z are in a bin. Note that the last bin might not be filled. In that case, those elements could be divided into the already existing bins, or simply discarded.
   - If $Z = \emptyset$, then take 2 elements from X (resp. 3 elements from Y) and form a bin with them. Repeat this process until X and Y have less than 2 and 3 elements, respectively. The remaining players could be divided into the existing bins or simply discarded too.

   Note that Condition 5 is satisfied for the bins filled at this step.

In [3] it is showed that the above algorithm has complexity $O(n \log^2 n)$, where n denotes the number of players subscribed.

To conclude this section, let us discuss the choice of the bin capacity C. As seen in the heuristic above, it seems reasonable to take $C = 1 + \max_{1 \leq i \leq n} w_i$. However, depending on how the data items are distributed among the players, this election might affect other conditions. As an example, suppose a campaign with MODL 1000 and the data owners that signed up as follows:

- The first has dataset length 500.
- The remaining 100 data owners have dataset size 5.

We quickly observe that this data distribution ends up with only two bins, the second one with 100 players (including the scientist).

In this situation, it seems reasonable to take C as the mean of dataset sizes. Note that, in this case, C will not be an upper bound for every w_i. Step 2 makes it an upper bound by assigning the players P_i with $w_i > C$ to different bins and completing these bins with other players.

In consequence, the data scientist should choose the bin capacity C according to the data distribution among the players. A bin capacity must be large enough to fulfill Condition 7, but small enough to satisfy Condition 3. Setting $C = 1 + \max_{1 \leq i \leq n} w_i$ is highly recommended in general, but distributions with few large outliers work better with C being for example the mean of the dataset sizes.

As we have seen, a good election of the bin capacity makes Conditions 3 and 7 be fulfilled. In conclusion, this algorithm seems to fit quite well with the grouping protocol we were aiming to find, see Conditions 1 through 7 above.

3.4 The Shamir Protocol

When the scientist has split the players into bins, a transaction is sent to the blockchain specifying them. Then, the campaign enters in the next phase. At this moment, data owners know the bin they belong to and have to gather all the necessary information to set up the SCALE-MAMBA network. They do this by making call transactions to the Smart Contract.

Once they have stored all the needed information, they send another transaction telling they are ready to start the computations. When all players from a bin have sent this special transaction, they head to SCALE-MAMBA and perform the SMPC execution using the Shamir Protocol. We hence have as many SMPC executions as bins in the campaign.

In this section we will focus on the SMPC execution of one of those bins. We will study the Shamir Protocol in detail, mainly concerned about making understandable for the reader how it works and why is it secure. The main references used for this purpose are [12] and [13]. For the rest of this section, we will work over a finite field $\mathbb{F}_p$ where p is a prime. We let the SMPC network have n players. We denote the set of players by $P = \{P_1, \ldots, P_n\}$. The players obtained the model to be trained as a circuit when making the call transactions above.

Section 2.1 considered an SMPC protocol with $n/2$ threshold condition. Threshold adversary models were extended to structure-based models in [14]. This generalization involves stronger results compared to threshold models, as it is typical in the literature.

3.4.1 Preliminaries

Structures. Let P be the set of all players, and let $\Gamma, \Delta \subseteq 2^P$, where 2^P denotes the set of all subsets of P. If $\Gamma \cap \Delta = \emptyset$, then we call the pair (Γ, Δ) an *access structure*. A set of players $Q \in \Gamma$ is said to be *qualified*, while a set of players $U \in \Delta$ is said to be *unqualified*. In the literature it is typical to consider that supersets of qualified sets are qualified, while subsets of unqualified subsets are unqualified. If $\Gamma = \Delta^c$, where Δ^c denotes the complement in 2^P of the set Δ, we say that (Γ, Δ) is *complete*. We assume our access structure is complete. Let:

- $\mathcal{U}$ be all sets in Δ which are maximal by inclusion. We let $\mathcal{U} = \{U_1, \ldots U_k\}$, and call $\mathcal{U}$ the set of *maximally unqualified subsets*.
- $\mathcal{Q} = \{Q_1, \ldots Q_k\}$, where $Q_i = P \setminus U_i \in \Gamma$ for $i \in \{1, \ldots, k\}$. Recall that, since (Γ, Δ) is complete, all the complements are qualified. Thus, we call $\mathcal{Q}$ the set of *minimally qualified subsets*.

Finally, we say (Γ, Δ) is Q_ℓ if there are no ℓ sets in Δ such that the union of these sets results in P, that is, there do not exist $U_1, U_2, \ldots, U_\ell \in \Delta$ such that

$$P = U_1 \cup U_2 \cup \cdots \cup U_\ell.$$

From now on, we assume our access structure to be Q_2.

Secret Sharing Let P be the set of all players. We next describe how a player may share a secret $s \in \mathbb{F}_p$ with some subset S of players in P. First, the player samples uniformly some values $s_1, \ldots, s_k \in \mathbb{F}_p$ such that

$$s = \sum_{i=1}^{k} s_i$$

and then sends each s_i to the players in S. More precisely, the secret sharing protocol is defined as follows:

Protocol LSSS

Suppose we have a set of players P with a (Γ, Δ) access structure where the maximally unqualified subsets and the minimally qualified subsets are as above (recall that the number of qualified sets is k). We assume there is a secret $s \in \mathbb{F}_p$ that a player whishes to share. Then the player does the following:

1. Sample uniformly $k-1$ elements $s_j \in \mathbb{F}_p$, with $j \in \{1, \ldots, k-1\}$.
2. Compute $s_k = s - \sum_{j=1}^{k-1} s_j$.
3. The secret s_j is assigned to the qualified set Q_j, for each $j = 1, \ldots, k$. As such, s_j is sent to all the players in Q_j.

We introduce now the following notations:

- When writing $[[s]]$ we mean the secret sharing of value s with respect to the above scheme. We call such $[[s]]$ the *shared secret*.
- When writing s_Q we are referring to the part of the secret s that was assigned to the set Q, for every $Q \in \mathcal{Q}$. We call s_Q a *share of* s, for every $Q \in \mathcal{Q}$. With this notation the following equality holds:
$$\sum_{Q \in \mathcal{Q}} s_Q = \sum_{j=1}^{k} s_j = s,$$
- With the notation above, $[[s]] = (A_{s,1}, \ldots, A_{s,n})$, where $A_{s,i} = \{s_Q \mid P_i \in Q \text{ for } Q \in \mathcal{Q}\}$, for $i \in \{1, \ldots, n\}$.
- The secret sharing above allows qualified sets to open $[[s]]$. Since the access structure given is assumed to be Q_2, we know that every $Q, Q' \in \mathcal{Q}$ where $Q' \neq Q$ satisfy $Q \cap Q' \neq \emptyset$. Hence, at least one player in Q received s_Q and $s_{Q'}$ for all $Q' \in \mathcal{Q} \setminus Q$, and thus players in Q can open $[[s]]$.
- We denote as LSSS.PRSS(s) the above LSSS protocol when s is an arbitrary secret. We also allow to call LSSS.PRSS() without specifying s, in which case a random value is selected as s, see [12] for further documentation.
- It is important, at this stage, to give some informal information about *opening* a shared secret $[[s]]$. Players open $[[s]]$ when they collaborate to obtain value s. This collaboration involves communication using some *channels*, where players can send their shares of $[[s]]$ to each other. We will see these protocols in detail in Section 3.4.2.

Operations on the shares For players to be able to evaluate the circuit given, some methods are needed for players to be able to operate on shared secrets. These methods are the ones that make SMPC an actually useful tool, since they allow players to make operations on shared secrets and output results without leaking information about their secret inputs.

Before describing them, though, we need to introduce some notation. First of all, note that each player knows only a certain number of shares s_Q, for $Q \in \mathcal{Q}$. A *public value* $c \in \mathbb{F}_p$ is not written inside brackets and is known by all players. When writing $[[s]] + c$ we are referring to a shared secret $[[y]]$ such that when opening $[[y]]$, we have $y = s + c$. A similar convention is adopted with the notation $[[s]] \cdot c$, this time resulting in the opened value $s \cdot c$.

The operations we are concerned with are the following.

1. Operation 1: Having $[[s]]$ and $[[t]]$, to compute $[[s + t]]$.
2. Operation 2: Having $[[s]]$ and a public $c \in \mathbb{F}_p$, to compute $[[s]] + c$.
3. Operation 3: Having $[[s]]$ and a public $c \in \mathbb{F}_p$, to compute $[[s]] \cdot c$.
4. Operation 4: Having $[[s]]$ and $[[t]]$, to compute $[[s \cdot t]]$.

Operations 1 and 3 are trivial. For the first one, observe that the players always receive the shares of the same qualified sets $Q \in \mathcal{Q}$, *i.e.*, P_i can add $s_Q + t_Q$ for every $Q \in \mathcal{Q}$ such that $P_i \in Q$, for $i \in \{1, \ldots, n\}$.

The second is solved analogously. Each player multiplies by c each share they know, without having to communicate with the other players. For Operation 2 we have the following protocol:

Protocol for Operation 2

Players start the protocol with $[[s]]$ and c. They want to output some $[[x]]$ such that when opening $[[x]]$ they have $x = s + c$.

1. A set $Q \in \mathcal{Q}$ is agreed.
2. The players in Q perform the operation $s_Q + c$. The other players do not perform any addition.
3. This way, we have a new shared secret $[[x]] = (A_{x,1}, \ldots, A_{x,n})$, where:

$$A_{x,i} = \begin{cases} A_{s,i} \text{ if } P_i \notin Q, \\ \{s_Q + c\} \cup \{s_{Q'} \mid P_i \in Q' \text{ for } Q' \in \mathcal{Q} \setminus Q\} \text{ if } P_i \in Q. \end{cases}$$

This way, if we open $[[x]]$ players can compute:

$$\sum_{Q \in \mathcal{Q}} x_Q = (s_Q + c) + \sum_{Q' \in \mathcal{Q} - Q} s_Q = c + \sum_{Q \in \mathcal{Q}} s_Q = c + s.$$

For Operation 4, recall that (Γ, Δ) is assumed to be Q_2. This means that for every $Q, Q' \in \mathcal{Q}$ with $Q \neq Q'$, there exists some player P_m satisfying $P_m \in Q \cap Q'$ for $m \in \{1, \ldots, n\}$. Hence P_m can compute: $s_Q t_{Q'}$, $s_{Q'} t_Q$, $s_Q t_Q$, $s_{Q'} t_{Q'}$. Let us observe that if $|Q \cap Q'| > 1$ then there is more than one player which is able to compute the previous values. In such situations, only one of the players in $Q \cap Q'$ is assigned those values, while the others simply skip them. This way, we assure that no value $s_Q t_{Q'}$ will be repeated in the multiplication protocol below, for every $Q, Q' \in \mathcal{Q}$ such that $Q \neq Q'$. Therefore, the players can together compute all the crossed terms of

$$s \cdot t = \left(\sum_{Q \in \mathcal{Q}} s_Q \right) \cdot \left(\sum_{Q \in \mathcal{Q}} t_Q \right).$$

In [14], Maurer introduced the following protocol for multiplying two shared secrets.

Protocol for the product (MAURER)

Let s and t be two secret values and let $[[s]]$ and $[[t]]$ be the corresponding shared secrets of s and t, respectively, among the players in P. Our aim is that all players obtain $s \cdot t$ without having additional information about s and t.

1. Split the set $\mathcal{T} = \{(i, j) \mid 1 \leq i, j \leq k\}$ into:
   - $\mathcal{T}_1 = \{(i, j) \mid P_1 \in Q_i \cap Q_j\}$
   - $\mathcal{T}_2 = \{(i, j) \mid P_2 \in Q_i \cap Q_j\}$
   - $\vdots$
   - $\mathcal{T}_n = \{(i, j) \mid P_n \in Q_i \cap Q_j\}$

   The sets $\mathcal{T}_1, \ldots, \mathcal{T}_n$ form a partition of $\mathcal{T}$, *i.e.*, each pair (i, j) is only assigned to one player, for $1 \leq i, j \leq k$. This way, we avoid the shares to be repeated.
2. Each player P_m with $m \in \{1, \ldots, n\}$ computes

$$v_m = \sum_{(i,j) \in \mathcal{T}_m} s_{Q_i} t_{Q_j}.$$

3. Each player P_m calls LSSS.PRSS(v_m) to secret share v_m, for $m \in \{1, \ldots, n\}$. At the end of this step we have the new created shared secrets $[[v_1]], \ldots, [[v_n]]$.
4. The parties, once they have received their shares of $[[v_1]], \ldots, [[v_n]]$ (say $A_{v_1,i}, \ldots, A_{v_n,i}$ for $i \in \{1, \ldots, n\}$), compute locally
$$[[z]] = \sum_{m=1}^{n} [[v_m]].$$
Note that the addition above corresponds to Operation 1.

Let us observe that, when opening $[[z]]$, the players for which the value z is opened can compute
$$z = \sum_{Q \in \mathcal{Q}} z_Q.$$
On the other hand, we know that
$$\sum_{m=1}^{n} v_m = \sum_{m=1}^{n} \sum_{(i,j) \in \mathcal{T}_m} s_{Q_i} t_{Q_j} = \sum_{(i,j) \in \mathcal{T}} s_{Q_i} t_{Q_j} = \sum_{Q \in \mathcal{Q}} s_Q \cdot \sum_{Q \in \mathcal{Q}} t_Q = s \cdot t.$$
Note that the third equality holds since (Γ, Δ) is Q_2. Then trivially
$$z = \sum_{m=1}^{n} v_m = s \cdot t.$$
The above protocol, thus, makes it possible for players to multiply two shared secrets. The method, however, needs communication among them, and is quite heavy to perform. For this reason, while the methods seen for Operations 1 to 3 are used in the online phase of the Shamir Protocol, this one cannot be used. It will instead be used in the offline phase. SMPC protocols tend to perform all heavy computations in the offline phase, in order to have only light computations in the online phase. But we still need a method to compute products in the online phase. For this purpose, the so called Beaver triples are essential, as they introduce a faster multiplicative method once Maurer's is done in the offline phase.

Beaver Triples Let $[[a]]$, $[[b]]$ be shared secrets. A Beaver triple is a triple
$$([[a]], [[b]], [[c]])$$
such that c satisfies $c = ab$. Now let us see the way players can use these triples in order to compute a product, without the need of Maurer's multiplication protocol.

Protocol for the product (TRIPLES)

Suppose players have two shared secrets $[[x]]$ and $[[y]]$, and a Beaver triple $([[a]], [[b]], [[c]])$. So as to compute $[[x \cdot y]]$, they do the following:

1. Compute:
   - $[[\alpha]] = [[x]] - [[a]]$.
   - $[[\gamma]] = [[y]] - [[b]]$.
2. Call the functions OPEN.Reveal($[[\alpha]], 0$) and OPEN.Reveal($[[\gamma]], 0$). Then all players have α and γ.
3. Set $[[z]] = [[c]] + \alpha \cdot [[b]] + \gamma \cdot [[a]] + \alpha \cdot \gamma$. Observe that computing $[[z]]$ only involves computationally cheap operations on the shares.

Then in shared secret $[[z]]$ we have:
$$[[z]] = [[a \cdot b + (x - a) \cdot b + (y - b) \cdot a + (x - a) \cdot (y - b)]]. \tag{2}$$
This formula comes from the fact that:

- $[[\alpha]] = [[x]] - [[a]] = [[x - a]]$.
- $[[\gamma]] = [[y]] - [[b]] = [[y - b]]$.
- $c = a \cdot b$.

Simplifying (2) we obtain the desired result:

$$[[z]] = [[x \cdot y]].$$

In conclusion, Beaver triples are computationally cheaper for computing products if it is needed in the circuit evaluation. However, they must be previously generated. This is the reason why we need a protocol to randomly generate them before the circuit evaluation occurs. This triple generation will be done using Maurer's protocol for the product, while the circuit evaluations that involve products in the online phase will use Beaver triples.

3.4.2 Complementary protocols in Shamir Protocol

We now introduce some auxiliary functions that will be needed in the following sections.We begin with a protocol involving *hash functions*. Recall that all the inputs and computations lie in a field $\mathbb{F}_p$ for some fixed prime p. Let N be the number of bits p has in base 2, let $H : \{0,1\}^N \longrightarrow \{0,1\}^M$ be a collision resistant hash function for some $M < N$, and let $v \in \mathbb{F}_p$. When computing $H(v)$ the following is implicitly happening:

- We write v in base 2. We let v in binary be v_2.
- Since $v \in \mathbb{F}_p$, v_2 must have $k \leq N$ bits. We add $N - k$ zeros to v_2, so that v_2 has exactly N bits.
- We compute $H(v_2)$.

Protocol HASH

Let $H : \{0,1\}^N \longrightarrow \{0,1\}^M$, be a collision resistant hash function for some $M < N$ as defined above, and let $v \in \mathbb{F}_p$.

- Init(): Set $v = 0$.
- Update(v'): Update $v = v + v'$ (without computing $H(v)$).
- Finalise(): Compute and output $H(v)$ following the steps explained before.

The second protocol we will see is the one that we will use to open the shared values. But first, we need to define a partition of the set $\mathcal{Q}$ as follows: consider all the maps $f : \mathcal{Q} \mapsto P$ such that for every $P_i \in P$, $f(Q) = P_i$ implies $P_i \in Q$. Then, choose f such that $\mathrm{Im}(f)$ is as large as possible. Fixing that f, let $\mathcal{Q}_i = f^{-1}(P_i)$ for each $P_i \in P$, where $f^{-1}(P_i)$ denotes the preimage of P_i under f. One can verify that the sets $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ form a partition of $\mathcal{Q}$.

We assume the existence of a partition $\{\mathcal{Q}_i\}_{i=1}^n$ where every $\mathcal{Q}_i$ is non-empty [4] from now on.

Protocol OPEN

Let $P = \{P_1, \ldots, P_n\}$ be the set of players and let $\{\mathcal{Q}_i\}_{i=1}^n$ be a partition of $\mathcal{Q}$.

- Init(): All players call HASH.Init().
- Broadcast(j, m):
  1. P_j sends m to all players over authenticated channels.

[4] See [12] for the discussion on the existence of partitions of $\mathcal{Q}$ with non-empty sets.

2. Each player executes HASH.Update(m).

- Reveal($[[s]], j$): This method is used to open a shared secret $[[s]]$. Recall that when opening $[[s]]$, the goal is to collect enough shares of $[[s]]$ so as to obtain s. There are two options for opening secrets: revealing a shared secret to only one player, or revealing a shared secret to all players.
  - If $j = 0$ then the secret will be opened to everyone. Note that since the partition $\{\mathcal{Q}_i\}_{i=1}^{k}$ is assumed to satisfy that all $\mathcal{Q}_i$ must be non-empty, we know that at least there exists some $Q \in \mathcal{Q}_i$:
    1. For every $P_i \in P$ and for all $Q \in \mathcal{Q}_i$, P_i sends s_Q to all players that are not in Q. They use authenticated channels. Now every player can compute $s = \sum_{Q \in \mathcal{Q}} s_Q$.
    2. Every player calls HASH.Update(s_Q) $\forall Q \in \mathcal{Q}$.
    3. The players call OPEN.CompareView().
  - If $j \neq 0$ then the secret will be opened only to P_j:
    1. For every $Q \in \mathcal{Q}$ such that $P_j \notin Q$, the players in Q send s_Q to P_j over a secure channel.
    2. If any two shares from the players of some set Q are different then P_j aborts the protocol. If not, P_j calculates
$$s = \sum_{Q \in \mathcal{Q}} s_Q.$$
- CompareView():
  1. Each $P_i \in P$ executes HASH.Finalise() and gets an output h_i.
  2. Each $P_i \in P$ sends h_i to the other players.
  3. Each player compares all hashes received and:
     - If all hashes are equal they do not abort and run HASH.Init().
     - Otherwise they abort and run HASH.Init().

3.4.3 The offline phase

As seen above, Beaver triples are computationally cheaper for computing products than Maurer's protocol. However, the triples must be previously generated. This is the reason why we need a protocol to randomly generate them before the circuit evaluation occurs. Such protocol is based on Maurer's protocol, which is computationally more expensive. As a consequence, there are two phases in the Shamir Protocol (as in most of SMPC protocols):

- The offline phase, where the Beaver triples and other items are randomly generated. The offline phase protocol in the Shamir Protocol is a generalization of Protocol 3.1 in [15] for n players, using Maurer's multiplication protocol.
- The online phase, where the circuit is evaluated.

This section is intended to explain Protocol 3.1 in [15]. For this protocol to be clear we need to give an informal explanation of it, and along the way introduce some useful notation:

- We will use n_T to denote the number of triples we want to output from the protocol.
- The idea of the protocol is to generate more triples than the n_T needed. This is because we will sacrifice the others in order to assure that the n_T needed are correct. Correctness means that each of the triples satisfies that, if such triple is $([[a]], [[b]], [[c]])$, then $c = a \cdot b$. We will use M to denote the total number of triples we will generate at the beginning of the process.
- We will use $\vec{D}$ to denote the array that contains all triples that we generate. During the protocol, this array will be split into K subarrays $\vec{D_1}, \dots \vec{D_K}$, and each subarray $\vec{D_k}$ will also be split into L sub-subarrays $\vec{D_{k,1}}, \dots, \vec{D_{k,L}}$. Those sub-subarrays will be denoted as $\vec{D_{k,i}}$ for $k \in \{1, \dots, K\}$ and $i \in \{1, \dots, L\}$. It is important to note that L is assumed to divide n_T.

- There will be a moment where those subarrays and sub-subarrays will be shuffled. We want the shuffling and validation to be as random as possible. This would mean that, when later opening the shares, if a player cheats when sending their triples, the randomness will allow the other players to know someone cheated. To make this possible we will need to *check* (*i.e.*, open and verify that the triple was correct) S triples per each $\vec{D_{k,i}}$, and to make those S triples per sub-subarray be as randomly chosen as possible. It is important to say that this *checking* is not a triple sacrifice.
- Once we have checked enough triples, we will divide the remaining ones into *buckets* of triples. We will require the previous steps to lead us to n_T buckets, each one of size K. We use $\vec{\beta}$ to denote the array consisting of all the buckets, and $\vec{\beta}_1, \ldots, \vec{\beta}_{n_T} \in \vec{\beta}$ to denote the buckets inside of $\vec{\beta}$, so $\vec{\beta} = \{\vec{\beta}_1, \ldots, \vec{\beta}_{n_T}\}$.
- There will be a moment inside the protocol where, per each bucket, $K-1$ triples will be used to check that one triple is correct. This process is usually called *sacrificing the triples* in the literature. At this stage, we will need some new notation for the triples. We let the triple $([[a_j^i]], [[b_j^i]], [[c_j^i]])$ be the j-th triple inside the i-th bucket, for $j \in \{1, \ldots, K\}$ and $i \in \{1, \ldots, n_T\}$.

The offline phase works as follows:

Protocol OFFLINE

On input n_T, we will first of all need to calculate M, for us to know how many triples we have to generate. In [15] it is shown that $M = (n_T + SL)(K-1) + n_T$.

1. For $j \in \{1, \ldots, M\}$ the players call LSSS.PRSS() (there is no input since we want the shares to be random) twice per round to obtain the pair of random shared secrets $([[a_j]], [[b_j]])$.
2. For each $([[a_j]], [[b_j]])$, all players call MAURER$([[a_j]], [[b_j]])$ and obtain the output $[[c_j]]$, where c_j is supposed to be $a_j b_j$. Let us observe that in this moment, each player has M randomly generated triples $([[a_j]], [[b_j]], [[c_j]])$.
3. We now have $\vec{D} = \{([[a_j]], [[b_j]], [[c_j]])\}_{j=1}^{M}$. Each player splits $\vec{D}$ into $\vec{D_1}, \ldots, \vec{D_K}$, where:
   - $\vec{D_1}$ has length n_T.
   - $\vec{D_2}, \ldots, \vec{D_K}$ each have length $n_T + SL$.
4. For $k \in \{2, \ldots, K\}$, each player splits $\vec{D_k}$ into L subarrays $\vec{D_{k,1}}, \ldots, \vec{D_{k,L}}$ (since L divides n_T) each of size $n_T/L + S$.
5. For $k \in \{2, \ldots, K\}$ and $i \in \{1, \ldots, L\}$ players jointly and securely shuffle (by this expression, we mean all the shufflings are computed locally but following the global order, so that no disorder happens after the shuffling) each triple in $\vec{D_{k,i}}$.
6. For $k \in \{2, \ldots, K\}$ players jointly and securely shuffle each $\vec{D_{k,i}}$ in $\vec{D_k}$.
7. For $k \in \{2, \ldots, K\}$ and $i \in \{1, \ldots, L\}$:
   (a) All players call OPEN.Reveal$([[c_s]], 0)$ where $s \in \{1, \ldots, S\}$ and S is referring to the first S triples in $\vec{D_{k,i}}$. Recall that we have $(n_T/L) + S$ triples per each $\vec{D_{k,i}}$.
   (b) If the protocol continues without abort, then all players remove the opened triples from each $\vec{D_{k,i}}$.
8. Note that now the length of each D_k is n_T for all $k \in \{1, \ldots, K\}$. Thus the length of $\vec{D}$ is $n_T K$. Now we do the following:
   - For each $\vec{D_k} = (t_1, \ldots t_{n_T})$ (where t_i denotes the i-th triple in $\vec{D_k}$) take t_i and add it to $\vec{\beta_i}$.
   - By doing the previous step, we have generated n_T buckets $\vec{\beta_1}, \ldots \vec{\beta_{n_T}}$, each one of size K.
   - For $i \in \{1, \ldots, n_T\}$ and for $j \in \{2, \ldots, K\}$ all parties do:
     (a) Call LSSS.PRSS() to output a random shared value $[[r]]$.
     (b) Call OPEN.Reveal$([[r]], 0)$ to output value r.
     (c) Call OPEN.Reveal$(\sigma, 0)$, OPEN.Reveal$(\tau, 0)$ where:

- $\sigma = [[b_1^i]] - [[b_j^i]]$ (can be computed locally, see 3.4.1)
- $\tau = r \cdot [[a_1^i]] - [[a_j^i]]$ (can be computed locally, see 3.4.1)

(d) Compute locally the value:

$$[[z^i]] = r \cdot [[c_1^i]] - \sigma \cdot [[a_j^i]] - \tau \cdot [[b_j^i]] - [[c_j^i]] - \sigma\tau.$$

(e) Call OPEN.CompareView() and if necessary abort. If not, call OPEN.Reveal($[[z^i]], 0$) and output z^i. If $z^i = 0$ continue, if not abort.

- If the process reached this point without aborting, that means that the first triple in each $\vec{\beta_i}$ has been properly validated and can be used for the online phase. As we have n_T buckets, we obtained succesfully n_T triples, as we wanted.

We only have one thing else to do before moving forward to the online phase. In the above protocol, we assumed that when opening $[[z]]$ we should have obtained $z = 0$. Let us see this. By definition:

$$[[z]] = r \cdot [[c_1^i]] - \sigma \cdot [[a_j^i]] - \tau \cdot [[b_j^i]] - [[c_j^i]] - \sigma\tau,$$

where

- $c_1^i = a_1^i \cdot b_1^i$
- $c_j^i = a_j^i \cdot b_j^i$
- $\sigma = [[b_1^i]] - [[b_j^i]]$
- $\tau = r \cdot [[a_1^i]] - [[a_j^i]]$

Operating on the shares and making those changes we have that:

$$[[z]] = [[r \cdot a_1^i \cdot b_1^i - (b_1^i - b_j^i) \cdot a_j^i - (r \cdot a_1^i - a_j^i) b_j^i - a_j^i \cdot b_j^i - (b_1^i - b_j^i)(r \cdot a_1^i - a_j^i)]].$$

And now simplifying we obtain the desired result:

$$[[z]] = [[0]].$$

3.4.4 The online phase

We are ready to study now the online phase of the Shamir Protocol. This protocol can be briefly explained as follows: the players perform an LSSS.PRSS()-based secret sharing protocol to share their inputs, then each player performs the necessary computations (following the given circuit) on the received shares, and finally those results are opened to obtain the output.

The online phase works as follows:

Protocol ONLINE

- Init(n_T):
  1. Every player calls HASH.Init().
  2. The players jointly call OFFLINE and obtain the multiplicative, square and bits triples needed. The process OFFLINE only generates the multiplicative triples, but can be generalized to the others.
- ShareInput(x, j): This function is called to create a share $[[x]]$, where x is the input of P_j.
  1. The players jointly call LSSS.PRSS() to obtain $[[r]]$, where r is a random value. Recall that LSSS.PRSS() was the secret sharing protocol.
  2. The players call OPEN.Reveal($[[r]], j$) to open $[[r]]$ for P_j. Now P_j knows r.
  3. P_j calculates $\epsilon = x - r$ and calls OPEN.Broadcast(j, ϵ) to send ϵ to all players.

4. Players compute $[[r]] + \epsilon$ to obtain $[[x]]$, which is a share of input x.

- Add($[[x]], [[y]]$): Function to add two shares. On input $[[x]], [[y]]$, each player locally computes $[[x]] + [[y]]$ to obtain a shared secret $[[z]]$ such that $[[z]] = [[x + y]]$.
- Multiply($[[x]], [[y]]$): Function to obtain the product of two shares.
  1. Take one unused triple from the ones generated in OFFLINE protocol. Suppose that the triple taken is $([[a]], [[b]], [[c]])$, where $c = a \cdot b$.
  2. Call TRIPLES($[[x]]$, $[[y]]$, ($[[a]]$, $[[b]]$, $[[c]]$)) to output a shared secret $[[z]]$ such that $[[z]] = [[x \cdot y]]$.
- Output($[[s]], j$): This function gives the output of a secret shared item $[[s]]$.
  1. The players run OPEN.CompareView(). If the protocol results in abort, the players abort the output protocol. If not, continue.
  2. Do the following:
     - If $j = 0$ the result is for all players:
       (a) The players jointly call OPEN.Reveal($[[s]], 0$) to open s.
     - If $j \neq 0$ the result is only for player P_j:
       (a) The players jointly call OPEN.Reveal($[[s]], j$).
       (b) If the protocol aborts, abort the output. If not, output s to P_j.

3.5 The payment process

Once players have finished the SMPC execution on their bin, each of them will have obtained one between two possible results: **SUCCESS** or **ABORT**. At this moment, every player will send a transaction to the Smart Contract telling the result obtained at the end of the execution. This process happens simultaneously in all bins, and the payments will work analogously in every bin, so from now on we assume we are in one of all those bins.

Intuitively, the payment process will work as follows: if most of the players in a bin obtain **SUCCESS** the payments are executed and the reputations increase, and if most of them obtain **ABORT** the payments are not executed and the reputations decrease.

Let the bin have n players (the scientist included). Throughout the paper we assumed a $t < n/2$ threshold, *i.e.*, we are assuming that more than $n/2$ players behave correctly. This assumption can be extended to reporting the output each player obtained in the SMPC execution. In other words, we assume that the transaction will be automatically sent during the SMPC protocol by each player, and thus only corrupted players can potentially report a different output from the one they received.

With the assumptions above, the payment process results in the following cases:

- More than $n/2$ players report **SUCCESS**:
  - Tokens are paid according to Section 3.2.
  - Reputations are increased. Recall from section 3.2 that a player with reputation R gambled part of their reputation $r < R$. In fact, from Section 3.2, r must be greater than the minimun reputation established by the scientist. If the reputations are increased, this player would receive $2r$ (the gambled reputation plus r).
- More than $n/2$ players report **ABORT**:
  - Tokens are returned to the data scientist.
  - Reputations are decreased, *i.e.*, the gambled reputations are not returned to the players (the scientist included).
- Exactly $n/2$ players report **SUCCESS** and the other $n/2$ report **ABORT**: In this unlikely situation, we assume that the data scientist is not corrupted. The data scientist's report will therefore decide whether the SMPC ended with **SUCCESS** or with **ABORT**.

4 Conclusions

Regarding what we have seen in Section 3, the cycle of a campaign introduced in Section 2.2.1 can be completed as follows:

1. A data scientist registers an SMPC request specifying:
   - The Minimum Overall Dataset Length.
   - The total amount of tokens they will pay to data owners.
   - The minimum reputation a data owner must put in stake to be able to subscribe.

   The request is included inside a new campaign that is published in the marketplace. This step can be skipped if the data scientist reuses an old campaign.
2. Data owners receive the campaign and if it fits to them they sign up for it by specifying:
   - The reputation they will stake (must be greater than the minimum required by the scientist).
   - The dataset size they will provide.
3. When overall dataset length surpasses the MODL, the subscribing time ends and the data scientist starts the grouping protocol. Once it ends, data owners are informed about the bin they belong to, and prepare themselves for the SMPC execution.
4. The SMPC execution is done independently and simultaneously in all bins.
5. During the SMPC execution, an automatically generated transaction is sent to the Smart Contract. This transaction reports if the SMPC execution ended with **SUCCESS** or with **ABORT**.
6. According to the reports of the previous step, the payments occur. If the majority in a bin happens to be **SUCCESS**, then payments are executed and the gambled reputations are increased. If it happens to be **ABORT**, then payments are not executed and reputations are decreased.
7. The campaign is marked as finished. This campaign can be reused.

4.1 Further work

As we have seen so far, the theoretical approximation to a private data marketplace given in this document is not simple at all. It is however complete and cares mostly about the security of private data and the correctness of the SMPC executions.

Nevertheless, this theoretical design does not fit exactly with the actual implementation of the marketplace. Indeed, there is work to be done:

- The grouping protocol has not been implemented, and thus only the theoretical solution is exposed in this paper.
- The transaction reporting whether an SMPC execution ended with **SUCCESS** or **ABORT** is not automatically generated in the implementation, so players have more chances to lie in their reports.
- Let us observe that the scientist will have to deal with as much SMPC executions as bins are created in the grouping protocol. If the number of bins is large, the scientist will need a lot of computation resources to overcome all the SMPC executions succesfully. This is an open problem to study in the future.
- Currently, the Ethereum Blockchain does not seem a good choice even if it gives us all of the properties described in Section 2.2. This has to do with the actually high market capitalization of the *ether* token nowadays. As a consequence, sending a transaction to register a new SMPC request can be so expensive for scientists that they may not have incentives to participate in the marketplace. Other public blockchains, such as Polkadot or Solana, may be considered in future revisions.
- As the implementation is still in development phase, the token used to pay for each model training has not value at all. As such, it can be directly obtained by calling a function in the Smart Contract. The idea is to remove that function with the first stable release of the marketplace.

CRediT authorship contribution statement

Julen Bernabé-Rodríguez: Methodology, Software, Formal analysis, Investigation, Writing – original draft, Writing – review & editing, Visualization.

Albert Garreta: Conceptualization, Validation, Formal analysis, Investigation, Writing – review & editing.

Oscar Lage: Conceptualization, Investigation, Writing – review & editing, Supervision.

Acknowledgements

The second named author was supported by the ERC grant PCG-336983, by the Basque Government grant IT974-16, by the Ministry of Economy, Industry and Competitiveness of the Spanish Government Grant MTM2017-86802-P. Additionally, the second named author was supported by the Basque Government through the BERC 2018-2021 program and by the Ministry of Science, Innovation and Universities: BCAM Severo Ochoa accreditation SEV-2017-0718.

References

[1] Assmann, S. F. (1983). *Problems in discrete applied mathematics* (Doctoral dissertation, Massachusetts Institute of Technology).

[2] Assmann, S. F., Johnson, D. S., Kleitman, D. J., & Leung, J. T. (1984). On a dual version of the one-dimensional bin packing problem. *Journal of algorithms*, 5(4), 502-525.

[3] Csirik, J. (1999). Two simple algorithms for bin covering. *Acta Cybernetica*, 14(1), 13-25.

[4] Csirik, J., Johnson, D. S., & Kenyon, C. (2001, January). Better approximation algorithms for bin covering. In *SODA* (Vol. 1, pp. 557-566).

[5] Jansen, K., & Solis-Oba, R. (2003). An asymptotic fully polynomial time approximation scheme for bin covering. *Theoretical Computer Science*, 306(1-3), 543-551.

[6] Rotaru, D. (2021): *Awesome MPC*. In his GithHub site. `https://github.com/rdragos/awesome-mpc`.

[7] Yao, A. C. C. (1986, October). How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)* (pp. 162-167). IEEE.

[8] Aly, A., Cong, K., Cozzo, D., Keller, M., Orsini, E., Rotaru, D., ... & Wood, T. (2021). Scale–mamba v1. 12: Documentation.

[9] Koch, K., Krenn, S., Pellegrino, D., & Ramacher, S. (2020, September). Privacy-preserving analytics for data markets using MPC. In *IFIP International Summer School on Privacy and Identity Management* (pp. 226-246). Springer, Cham.

[10] Eyal, I., Shlomovits, O., Manuskin, A., & Kolegov, D. (2020): *White-City: A Framework For Massive MPC with Partial Synchrony and Partially Authenticated Channels.* In their GitHub site. `https://github.com/ZenGo-X/white-city/blob/master/White-City-Report/white_city.pdf`

[11] Zhang, D., Su, A., Xu, F., & Chen, J. (2018). ARPA Whitepaper. *arXiv preprint arXiv:1812.05820.*

[12] Keller, M., Rotaru, D., Smart, N. P., & Wood, T. (2018, September). Reducing communication channels in MPC. In *International Conference on Security and Cryptography for Networks* (pp. 181-199). Springer, Cham.

[13] Smart, N. P., & Wood, T. (2019, March). Error detection in monotone span programs with application to communication-efficient multi-party computation. In *Cryptographers' Track at the RSA Conference* (pp. 210-229). Springer, Cham.

[14] Maurer, U. (2006). Secure multi-party computation made simple. *Discrete Applied Mathematics, 154(2)*, 370-381.

[15] Araki, T., Barak, A., Furukawa, J., Lichter, T., Lindell, Y., Nof, A., ... & Weinstein, O. (2017, May). Optimized honest-majority MPC for malicious adversaries—Breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy (SP)* (pp. 843-862). IEEE.

[16] OpenZeppelin Community (2021): ERC20 Smart Contract, on their Github Repository. `https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts/token/ERC20`

[17] IPFS Community (2021): Main Webpage. https://ipfs.io/

[18] Williams, C. (2021): Eventeum, Github Repository. https://github.com/eventeum/eventeum

[19] Bernabé, J. (2021): SMPC and Blockchain, Github Repository. https://github.com/julenbernabe/SMPC-BCK