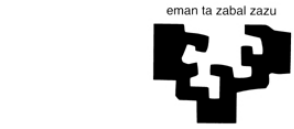

ZTF-FCT
Zientzia eta Teknologia Fakultatea
Facultad de Ciencia y Tecnología

SMPC & BLOCKCHAIN: Creating Private Data Marketplaces

Final Master Project
Master on Blockchain Technology
and Cryptoeconomics

Julen Bernabé Rodríguez

Work led by
Albert Garreta Fontelles
Oscar Lage Serrano

Leioa, September 13, 2021

Contents

Introduction

Data science is unstoppably evolving the last years. The main reason for that is the business behind those huge amounts of data that suddenly seem to be very useful for most of the companies around the world. No matter the size of the enterprises, all of them need to learn from their users' data so as to give a better service.

This need, however, knocks against the regulation that ensures the users' privacy. This means that companies actually cannot exploit their datasets as much as they would like to. Furthermore, they cannot share their results, or sell them, to other companies. This is a major drawback mainly for minor companies, since they do not have enough volume to benefit from their own data.

But times are changing. Some new technologies are tackling this issue, such as Homomorphic Encryption and Secure Multi-Party Computation. In a few years, these technologies will allow companies to learn from their direct competitors without leaking private information of their users. We might even see enterprises selling models that were trained jointly with other companies. That is a not so far future that may become reality sooner or later: private data marketplaces.

In this work, such a marketplace is introduced. Based on Secure Multi-Party Computation, we present a peer-to-peer marketplace that allows companies to jointly train a model with their private datasets. This marketplace is built upon blockchain technology, which gives it some very interesting properties. Chapter 1 gives a general overview of the marketplace, whereas Chapter 2 introduces the cycle of the Smart Contract deploying it. Chapter 3 gives a more detailed view of the interactions users must do with the Smart Contract so as to train a model. Chapter 4 studies Secure Multi-Party Computation in detail, and Chapter 5 introduces the economy behind the marketplace. Finally, a summary of the whole protocol is given in Chapter 6.

Chapter 1

A general overview.

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 scientist wants to meet data owners who can make their model better, while data owners want to take some profit from their private data, while assuring that information from their data will never be leaked.

1.1 Architecture of the solution.

We thus have two main components. On the one hand, we need a platform where data scientists and data owners can meet each other. There, data scientists can make model training requests, whom data owners might be listening to and can sign up for.

On the other hand, we need a technology that allows the data scientist and data owners to train a model while preserving data privacy. This tool outputs the result of the training to the data scientist.

1.1.1 SMPC and SCALE-MAMBA.

Secure Multi-Party Computation (SMPC) allows for a set $P = \{P_1, \ldots, P_n\}$ of n players to perform an arbitrary computation (or, equivalently, evaluate a given circuit) using the private inputs of the players, even if an adversary might corrupt some of them. The goal is that players learn the output, while no information but the one derived from such output is leaked.

In consequence, SMPC is a technology that suits well our use case. The data scientist and data owners can use SMPC for training the data scientist's model, while preserving the data owners' inputs private.

There are two main categories of SMPC protocols: the ones based on garbled-circuits (GC) and those based on Linear Secret Sharing Schemes (LSSS).

The LSSS-based protocol, as we will see later, uses a protocol to split secrets into so-called *shares* and then:

(i) The shares are distributed among the players. Each player has a circuit they have to evaluate.

(ii) Each player performs the computations on the shares received from the previous step following their circuit.

(iii) The final shares from step 2 are combined to reconstruct the output.

Most LSSS-based SMPC protocols tend to divide 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.

Currently, there are several SMPC implementation projects in course [16]. In the present work we use SCALE-MAMBA due to its long history in the filed and to the advanced state of development of its software. SCALE-MAMBA supports four instances of SMPC protocols:

- Full-Threshold.
- Shamir sharing.
- Replicated sharing.
- General $\mathsf{Q}_2 - MSP$ sharing.

Among the above, we use the Shamir sharing setting due to its simplicity. Since this protocol is LSSS-based, we will consider LSSS-based SMPC protocols in this document. These protocols will be simply called SMPC protocols from now on. Moreover, we will refer to the Shamir sharing protocol in SCALE-MAMBA as the Shamir Protocol. The main documentation and a summary of this can be found in [7].

The Shamir Protocol succeeds if and only if at least $n/2$ players behave correctly. If this condition is not fulfilled, the SMPC ends with an **ABORT** flag. In SMPC literature, these conditions are called *threshold conditions*. The $n/2$ threshold condition in the Shamir Protocol will be considered throughout the paper.

1.1.2 Why blockchain?

As mentioned in Section 1.1.1, 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, these technologies have a major drawback: once a network of players is set up, it is not possible to add or remove players. Suppose we only use the SCALE-MAMBA tool for our marketplace. Then scientists that 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 new players to the SCALE-MAMBA network 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, *i.e.*, data scientists and data owners. In addition, we need this platform to be suitable for processing payments.

More precisely, we want this platform to be:

- Open: anyone can access this platform.
- Borderless: users can access this platform wherever they are.
- Neutral: the key is the dataset value of the users, regardless of who they are.
- Censorship-resistant: there is no possibility of banning people in the platform.
- Public: everyone in the platform can verify that someone participated in previous model training requests. This way, players will have a reputation that can be key for future model trainings.
- Payment-suited: we want payments to be natural in the platform. These payments must satisfy the previous conditions too.

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.

1.2 Contributions.

In conclusion, integrating SMPC and blockchain technology sounds as a good option for building our marketplace. We intuitively consider the following protocol for such a marketplace:

(i) A data scientist makes an SMPC request on the blockchain.

(ii) All data providers receive the request and decide if they sign up or not. Those that want to participate sign up for the request on the blockchain.

(iii) After some time, the scientist ends the subscribing period and decides which data providers will participate in the SMPC computations.

(iv) The selected players perform the SMPC by using the Shamir Protocol.

(v) Data owners that performed the SMPC are paid for their services.

(vi) The process is repeated again.

This process can be seen in the figure below:

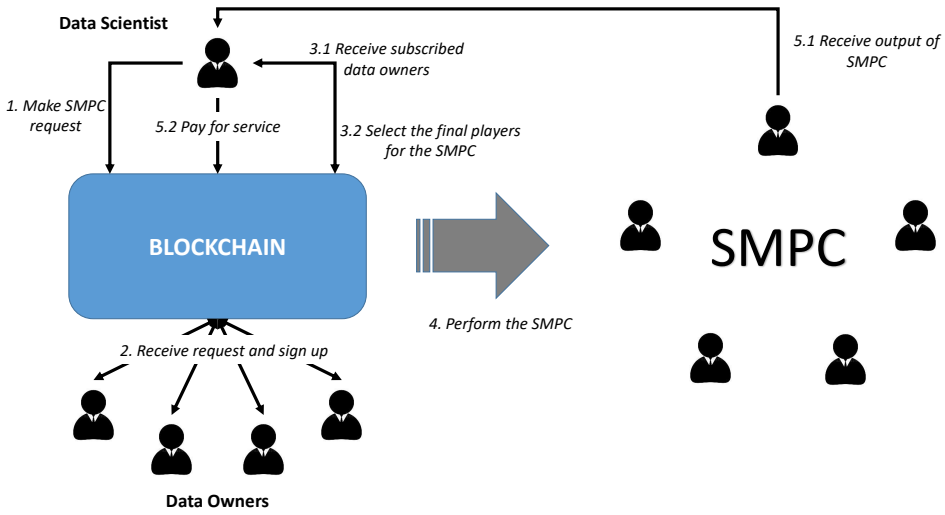

Figure 1.1: Overview of the protocol.

This work presents the architecture and implementation of a protocol as the above one. We assume that data scientists and data owners participating

in the marketplace have enough resources to perform the SMPC by themselves. This usually happens, for instance, in industrial environments. We also assume that at least $n/2$ players behave correctly during the SMPC, according to the Shamir Protocol in [7]. Thus, we only consider Byzantine Faults in this paper. 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 the present work with respect to other related work is the level of decentralization attained. Other works make different approaches to the solution, mainly assuming there are some computation nodes taking over the SMPC. In White-City project [23], the marketplace is based on a blockchain with several validator nodes that rule the network. This automatically implies more scalability, and data owners and data scientists do not need to mind about the computations. However, this marketplace is not open, nor censorship-resistant. The validator nodes might decide to ban some users by their own criteria. Moreover, the method used for rewarding the nodes for their work is not trivial at all. They hence do not give any incentive proposal for their project.

The ARPA project [24] is another approach for the marketplace. They use proxy Smart Contracts to connect the ARPA chain (their SMPC network) with an existing blockchain. In the ARPA chain, the SMPC nodes perform the computations. Consequently, this could be a more decentralized approach for the marketplace. However, their current implementation involves a coordinator that has some key functions inside the network. Thus, the network becomes centralized around the figure of that coordinator. Besides, we have again many actors to pay in this network. The incentives result in an unsolved problem in their protocol.

As a last approach, we have the KRAKEN project [22]. In KRAKEN, apart from the data scientists and data owners, we have the SMPC nodes. First of all, the data scientist publishes a model they want to train, and decides what data owners will participate in the SMPC. Those data owners send their datasets (cryptographically protected) to the SMPC nodes. The latter take upon the SMPC, and give directly the results to the data scientists. It is important to say that, in KRAKEN, all participants are authenticated. In consequence, this approach is not open. Furthermore, the incentives remain to be unsolved, due to the SMPC nodes that must be paid too.

With the above assumptions, this work gives the first trully decentralized approach to a private data marketplace. We besides give a grouping protocol based on the Bin Covering Problem that reduces Byzantine Faults. We

finally introduce a payment protocol based on the incentives that parties have when participating in the marketplace.

Chapter 2

The Smart Contract.

The characteristics of blockchain technology make it perfect for setting up our private data marketplace. Nevertheless, not all blockchains cover our needs. In fact, we want our blockchain to be:

(i) Open.

(ii) Borderless.

(iii) Neutral.

(iv) Censorship-resistant.

(v) Public.

(vi) Turing-complete.

The last item has to do with the computational properties of the blockchain. In 1936, Alan Turing created a mathematical model defining what he called a *Turing machine*. This machine consists of a state machine that manages symbols by reading and writing them on a sequential memory. This sequential memory was understood by him as a paper of infinite length.

Alan Turing also defined a system to be *Turing-complete* if it can be used to simulate any Turing machine. Such a system, therefore, can compute any algorithm that a Turing machine is able to compute, having limitations of finite memory.

2.1 A brief introduction to Ethereum and Smart Contracts.

The term *blockchain* came after the Bitcoin network. Bitcoin was firstly published in 2008 in [1]. This work combined different previous results to create the first completely decentralized digital cash system which does not depend on central authorities. This result was possible thanks to the Proof-of-Work concept, that brought up a solution for achieving consensus in decentralized systems. This solution for the consensus, briefly, consisted of a global election of blocks of transactions. These blocks are linked together forming a chain, the so-called *blockchain*.

In consequence, Bitcoin's blockchain perfectly tracks the current state of the bitcoins' ownership. When a block containing a 2 bitcoin transaction from user A to user B is accepted by the network, then the ownership of those 2 bitcoins has changed. Thus, the blockchain works as a distributed ledger.

Unfortunately, the Bitcoin network tracks only the state of currency ownership and, as such, is not intended to collect other kinds of data. This is the reason of the birth of Ethereum: creating a blockchain that can track any kind of data expressible as a *key-value tuple*. This makes Ethereum able to store data and code into its blockchain, and able to run that code and store the resulting changes too. Therefore, Ethereum can be understood as a decentralized Turing-complete system.

As said before, the Ethereum blockchain stores both data and code, and is able to run that code and store the results of the execution. 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 covers all we need for setting up our marketplace, since it is open, borderless, neutral, censorship-resistant, public and Turing-complete. Ethereum, in fact, allows us to create a Smart Contract deploying our marketplace, thus transfering to the marketplace all the properties mentioned before.

2.2 Overview of the Smart Contract.

The designed Smart Contract deploying the private data 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 Smart Contract works in a similar way. When a data scientist publishes an SMPC request, this request is added to the end of the to-do-list. Once the data owners are subscribed, the SMPC is executed, the result is published in the marketplace and the request is marked as finished in the to-do-list. Moreover, finished requests can be reused.

In more detail, the Smart Contract works as follows:

- A data scientist sends a transaction to the Smart Contract in order to make a new SMPC request or to reuse a finished one.
- Data owners who want to participate in the SMPC send a transaction to the Smart Contract to sign up for the request.
- When enough data owners are subscribed to the request, the data scientist sends a transaction to the Smart Contract specifying what data owners are finally selected to perform the SMPC.
- At this stage, the selected players do some call transactions, so as to store all the necessary information to set up the SCALE-MAMBA network. Once they collect all the needed data, they send a transaction to the Smart Contract telling they are ready to perform the SMPC.
- When all selected players have sent their ('ready') transactions, all players head to SCALE-MAMBA and perform the SMPC.
- The moment the SMPC is finished, the players automatically send a transaction to the Smart Contract telling the final state of the SMPC. Then, payments are performed.
- This request is considered finished until another scientist reuses it.

This process can be seen in the following figure. You can see the implementation of this Smart Contract in A. You can either refer to [28] to see the whole implementation.

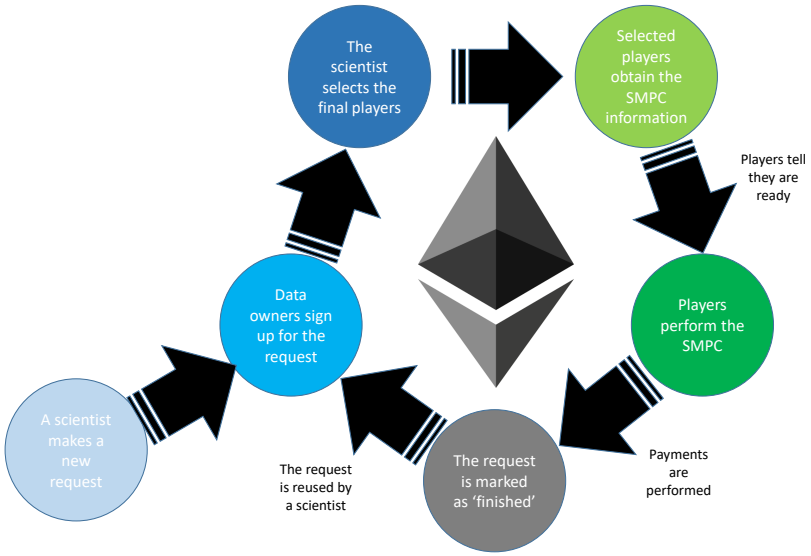

Figure 2.1: The cycle of a request.

Chapter 3

Handling SMPC requests on the marketplace.

In this section we will give exact descriptions of the conditions needed for obtaining security for the whole SMPC request process.

Before going ahead, it is important to say that the main idea to make this process secure is to 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 protocol with them.

3.1 Making a request.

In our process, the first to move is the data scientist by publishing an SMPC request. Then the data owners have the choice to sign up for it. However, the data scientist needs to establish some conditions to their request. For instance, when training a model, the size of the dataset is of vital importance for the data scientist. It is not the same to compute a mean among 9 salaries, than a mean among 10000 salaries. Considering this fact, it seems reasonable to ask the scientist about the *Minimum Overall Dataset Length* (MODL) they require for the model they want to train.

In addition, the scientist should tell the amount of money they will pay if computations finish succesfully. This information and the scientist's reputation are key for data owners to decide whether they sign up for the request or not. A well paid job is obviously more atractive for the data owners. Besides, more complex models (or models that require large 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 requests have some computing reputation. In other words, if the data scientist wants to train complex models, they will select only the data owners that have enough previous reputation. But recall that, at this stage, data owners are not subscribed yet. Thus, to prevent players with low reputations to sign up for the SMPC request, the scientist might establish a *minimum required reputation*. If a data owner does not reach this minimum reputation, they will not be able to sign up for the SMPC request. Summarizing, when registering an SMPC request, the data scientist has to establish the following:

- 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.
- The minimum reputation required for a data owner to sign up.
- The part of his reputation the scientist will gamble in this SMPC.

3.2 Signing up for an SMPC request.

Once the request is registered, data owners can see it and decide if they sign up or not for it. The minimum reputation required above hampers Byzantine Faults or OoS errors.

On the other hand, suppose a data owner receives an SMPC request and has enough reputation to sign up for it. Let that SMPC request have MODL $\mathcal{L}$ and total payment $\mathcal{P}$, and let w be the size of the data owner's dataset. Each data item will be paid, approximately, with $\mathcal{P}/\mathcal{L}$ tokens. Then, the data owner will receive $\hat{\rho}$ tokens, where $\hat{\rho} \in (\rho/2, \rho)$ and

$$\rho = \frac{\mathcal{P}}{\mathcal{L}} \cdot w. \tag{3.1}$$

In consequence, ρ gives a hint about the amount of tokens the data owner will receive for their services. We further discuss this in Section 5.1. As ρ is known at the subscribing time, a data owner can decide whether to sign up for a SMPC request based on it. We can conclude that it will be important for scientists to establish $\mathcal{P}$ and $\mathcal{L}$ correctly for the players to consider their SMPC requests to be well paid jobs.

In order to assure that $\hat{\rho} \in (\rho/2, \rho)$, we require the size of each data owner's dataset 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 an SMPC request they think is well paid. Suppose they have enough reputation to subscribe to it, but the scientist has no good reputation. If data owners do not consider this fact, then they might not be paid even if they perform the SMPC correctly[1]. As a conclusion, we can expect data owners to sign up only for well paid jobs from scientists with sufficiently good reputation. Of course, scientists might play with these requirements: a scientist with better reputation could pay worse for a job, and vice versa.

As a summary we obtain the following requirements for a data owner to sign up for an SMPC request at this stage:

- They must reach the minimum required reputation established by the scientist. Data owners will gamble that reputation[2].
- Their dataset must be of size $w < \mathcal{L}$.
- The SMPC request is a well paid job, 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. We will see this in Section 3.3.

3.3 The grouping protocol.

This section gives a full description of the way a scientist might select the players that will participate in the SMPC, once the data owners have signed up for the request. Recall that in SCALE-MAMBA, if a player suffers from a Byzantine Fault during the SMPC, all players will abort the SMPC protocol. Thus, splitting all players into subgroups reduces the probability of obtaining aborts. This way, we obtain as many outputs as subgroups we make. Hence, one player suffering from a Byzantine Fault would only output an abort in one subgroup, not in all of them. With this in mind, we give a grouping protocol that maximizes the number of subgroups to make. This, combined with the fact that all subscribed data owners reach the minimum reputation, reduces the chances of our SMPC computations failing.

[1] In future sections we will see why this can happen.

[2] They can gamble more reputation than the minimum one.

We introduce the following notation for the present section. Let n be the number of players that signed up for the request of the data scientist. We will denote the data scientist as P_0. Similarly, we will denote $P_1, \ldots, P_n$ the rest of players (data owners). We also let w_i be the size of the database of P_i. These database lengths will also be called *weights* from now on. Finally, we will refer to the aforementioned subgroups as *bins* from now on.

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

- ***Condition 1:*** *Data providers with large databases must end up assigned to some bin.* Observe that a data provider with a larger database w_i is more interesting to the data scientist.
- ***Condition 2:*** *Data providers with bad reputations will not be admitted.* A data provider that has succesfully participated in SMPC computations before is less likely to suffer from a Byzantine Fault.
- ***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 Byzantine Faults. 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.
- ***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 data number.* 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

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

The goal 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 element $s(e_i)$ as w_i. Then, when maximizing m we fulfill Condition 6. Condition 4 is straightforward. To satisfy Condition 5, the capacity of the bins C must be specified. We will discuss this later in this section, and assume an arbitrary C such that $1 \leq C < \mathcal{L}$.

Condition 2 is straightforward from Section 3.1: the scientist requires a reputation threshold to participate in the computations, and thus those players with not enough reputation will never be able to sign up for the SMPC. Furthermore, when giving an algorithm for the Bin Covering Problem, we will see that Condition 1 will be satisfied too.

The Bin Covering Problem was firstly studied by Assmann in [17], and Assmann *et al.* showed in [18] 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 [19]. Those algorithms are quite simple and practical, and they give a good approximation of the optimal result. In [20] and [21] other algorithms improving the approximation algorithms in [19] 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 [19] to group the players into bins.

In the protocol below, when assigning a player to a specific bin, we will say *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 used below. These boxes represent classes in an object-oriented programming language. The grouping protocol, thus, is the following:

Protocol GROUPING

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

(i) 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.

(ii) 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 assure that we have at least two players per bin.

(iii) 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 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 $Z = \{\hat{z}_1, \ldots, \hat{z}_{k_3}\}$. In these three sublists, some reordering might have happened: 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, as we will see below.

(iv) 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 place 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 placing them in the bin. If the elements in Z are not enough to fill the bin, take an element from Y to fill it.

(v) We have two cases:

- If $X \cup Y = \emptyset$, then do the following: 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.

As we can see from the algorithm described above, data providers with large databases will always be elected in the algorithm, and thus Condition 1 is satisfied. With this condition fulfilled too, this algorithm seems to fit quite well with the grouping protocol we were aiming to find, see Conditions 1 through 6 above. In [19], 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 request 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.

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 each dataset size 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 chooses the bin capacity C according to the data distribution among the players. 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.

Chapter 4

The SMPC protocol.

In this chapter we give an explanation of the Shamir Protocol inside the SCALE-MAMBA software. 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 $P = \{P_1, \ldots, P_n\}$. Recall that Section 1.1.1 considered an LSSS-based protocol with $n/2$ threshold condition.

4.1 Preliminaries.

4.1.1 Security of an SMPC.

When talking about the security of an SMPC protocol, we need to specify the type of adversary we will consider. Adversaries are usually defined by determining two aspects: the players the adversary can corrupt, and the way the adversary can corrupt them. The first aspect is studied in Sections 4.1.1.1 and 4.1.1.3. For the latter, we will consider two types of adversaries:

- **Passive adversaries:** They learn all the information of the players they corrupt, but those players always follow the protocol.
- **Active adversaries:** They take full control of those corrupted players. Hence the corrupted players can behave incorrectly.

4.1.1.1 *Threshold* security.

Threshold security studies the cases where an adversary can corrupt some players of the network, regardless of who they are.

Definition 4.1. In a network with n players, the adversary's corruption capability is set by a **threshold t** (with $t < n$) where it is assumed that the adversary can corrupt up to t players.

Under these assumptions it seems trivial that, in order to protect our secrets, we would need to set up a network where our secrets could be opened only if the number of players is strictly greater than t.

Observation 4.2. We quickly notice that the most secure network is the one whose threshold is $t = n - 1$. These protocols are called Full-Threshold protocols in the literature.

4.1.1.2 Structures. General notions.

Structures are a generalization of threshold-based SMPC protocols, as we will see later.

Definition 4.3. Let $S = \{S_1, \ldots, S_n\}$ be a finite set. We call

$$2^S = \{\emptyset, \{S_1\}, \ldots, \{S_n\}, \{S_1, S_2\}, \ldots, \{S_1, \ldots, S_n\}\}$$

the set of all subsets of S. This set is usually called the **power set of S**.

Definition 4.4. Let S be a finite set. A subset Π of 2^S is called a **structure** for S if Π is closed under taking subsets:

$$\forall S \in \Pi \text{ and } \forall S' \subseteq S : \ S' \in \Pi.$$

Example 4.5. Let us see the following two examples:

- Suppose $S = \{1, 2, 3\}$. Then $\Pi = \{\{1, 2\}, \{1, 3\}\} \in 2^S$ is not a structure for S since:
  - $\{1, 2\} \in \Pi$, and
  - $\{1\} \subseteq \{1, 2\}$,

  but $\{1\} \notin \Pi$.
- Suppose $S = \{S_i\}_{i=1}^n$ and $\Pi = \{T \subseteq 2^S \mid |T| < n\}$. Then Π is a structure since:
  - If $T \in \Pi$, then $|T| < n$
  - If $T' \subseteq T$, then $|T'| \leq |T| < n$.

  Hence, $T' \in \Pi$, and so Π is a structure.

From the second example we can deduce that, if the adversary's corruption capability is set by a threshold t in our SMPC protocol, then there is a structure of P containing all the subsets of P that can be corrupted by the adversary.

Structures allow to generalize the notion of *number of players the adversary can corrupt*. If we describe the subsets of players an adversary might corrupt by using structures, we can give a more specific approach of the adversary's capability to corrupt players. As an example, suppose a set of players $P = \{P_1, P_2, P_3, P_4\}$ where the adversary can only corrupt P_1 and P_3. Then, all possibilities of corrupting players are:

$$\Pi_1 = \{\{P_1\}, \{P_3\}, \{P_1, P_3\}\}$$

We can see that Π_1 is a structure, and thus we can specify perfectly the sets of players the adversary could corrupt. If conversely, we used the notion of a threshold, then we must have fixed $t = 2$, and then we are assuming that the adversary might corrupt the players in any of the following sets:

$$\Pi_2 = \{\{P_1\}, \ldots, \{P_4\}, \{P_1, P_2\}, \ldots, \{P_3, P_4\}\}.$$

In consequence, our approach is more general. This is one of the main reasons why we will describe the adversary's corrupting capabilities using structures.

Definition 4.6. Let S be a finite set. A subset $\Pi \subseteq 2^S$ is called an **anti-structure** for S if Π is closed under taking supersets:

$$\forall S \in \Pi \text{ and } \forall S' \in 2^S \text{ such that } S \subseteq S' : \ S' \in \Pi.$$

Observation 4.7. Every structure (resp. anti-structure) has some maximal subsets (resp. minimal subsets) inside that describe the whole structure (resp. anti-structure). From now on, structures and anti-structures will be described by those special sets.

Definition 4.8. Let Π_1 and Π_2 be two structures for a set S. Then we define the following operation:

$$\Pi_1 \sqcup \Pi_2 = \{S_1 \cup S_2 \mid S_1 \in \Pi_1 \text{ y } S_2 \in \Pi_2\}.$$

4.1.1.3 Structures for SMPC.

With all the notation and results from Section 4.1.1.2, we are now ready to see the use of structures for SMPC protocols. Recall that, when doing SMPC, each player has some private inputs (the private data) that will be used to perform the computations. Throughout the paper, we will denote such inputs as *secrets*, and we let them be in $\mathbb{Z}_p$.

Definition 4.9. Let P be the set of players and $P_j \in P$. We say that P_j **shares a secret** $s \in \mathcal{D}$ if P_j chooses $s_1, \ldots, s_k \in \mathbb{Z}_p$ such that

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

and sends each s_i to some subset of players in P[1].

Definition 4.10. Let P be the set of players, and suppose we want to share some secret s with the network. Then, there exist two types of subsets in P:

- **Qualified subsets:** Subsets of players that can reconstruct s. The collection of all qualified subsets is

$$\Gamma = \{Q \in 2^P \mid Q \text{ is qualified.}\}.$$

- **Unqualified subsets:** Subsets of players that cannot reconstruct s. The collection of all unqualified subsets is

$$\Delta = \{U \in 2^P \mid U \text{ is unqualified.}\}.$$

Lemma 4.11. *Γ and Δ from Definition 4.10 are respectively an anti-structure and a structure.*

Proof. Both results follow directly from Definitions 4.4, 4.6 and 4.10. □

In the literature, Δ is usually called **the adversary structure** for P.

Definition 4.12. Let P be the set of players, and let $\Gamma, \Delta \in 2^P$ as defined above. If $\Gamma \cap \Delta = \emptyset$, then we call the pair (Γ, Δ) an **access structure**. In addition, if $\Gamma = \Delta^C$, we say that (Γ, Δ) is **complete**.

Let Γ and Δ be as in Definition 4.10. We will assume throughout the paper that we have a complete access structure for our SMPC. As we saw in Observation 4.7, structures can be described by their maximal subsets. We 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**.

[1] We will return to this definition in Section 4.1.2. We now only need a rough definition to understand what we are doing.

We fix the above notation for the rest of the paper. We give now the last definition of this section:

Definition 4.13. Let P be a set of players and (Γ, Δ) a complete access structure for P. We say (Γ, Δ) is Q_ℓ if there are no ℓ sets in Δ such that the union of them 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.

4.1.1.4 Some security results.

Recall that, in an SMPC protocol, the adversary is described by a corruption type and by the number or set of players it can corrupt. In this section, the main conditions for achieving SMPC are given, assuming a specific description of the adversary. All proofs are omitted since they are all recorded in [3] and, in special cases where they are not, other articles will be mentioned.

Theorem 4.14. *In a network with n players where the adversary is passive (resp. active), SMPC is possible if and only if the threshold condition is $t < n$ (resp. $t < n/2$). This security is based on cryptographic intractability assumptions.*

Observation 4.15. Despite this result is mentioned in [3], in [5],[13],[14],[15] some SMPC protocols for Full-Threshold access structures (i.e. $t < n$), with active adversaries are shown. But the theorem remains being true since this protocols work with abort.

Theorem 4.16. *In a network with n players where the adversary is passive (resp. active) and its adversary structure is Δ, secure SMPC is possible if and only if (Γ, Δ) is Q_2 (resp. Q_3).*

Observation 4.17. In [3] specific results for mixed (active and passive) adversaries are given. They are beyond the scope of this work, but are another reason why structures are more suitable for describing the adversary's ability to corrupt players than thresholds.

4.1.2 Linear Secret Sharing Schemes.

In this section, the main results about Linear Secret Sharing Schemes (LSSS) are given. Recall that, when considering LSSS-based SMPC protocols, we need some protocol to share the secrets among the players.

In Section 4.1.2.1, we describe exactly the LSSS protocol used in the Shamir Protocol. Sections 4.1.2.2 and 4.1.2.3, in contrast, give descriptions about

operations on the shares, which will be crucial in both the offline and online phases of the Shamir Protocol.

4.1.2.1 The Linear Secret Sharing Scheme protocol.

Let P be the set of all players and $P_0 \in P$. We next describe how P_0 may share a secret $s \in \mathbb{F}_p$ with some subset S of players in P. Recall that, when a player shares a secret, we want that the players of $U_i \in \mathcal{U}$ cannot reconstruct the secret, for all $U_i \in \mathcal{U}$. First, P_0 samples uniformly some values $s_1, \ldots, s_k \in \mathbb{F}_p$ such that

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

and then P_0 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 P_0 whishes to share. Then P_0 does the following:

(i) Sample uniformly $k-1$ elements $s_j \in \mathbb{F}_p$, with $j \in \{1, \ldots, k-1\}$.

(ii) Compute $s_k = s - \sum_{i=1}^{k-1} s_i$.

(iii) 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*. Indeed, $[[s]] = (A_{s,1}, \ldots, A_{s,n})$, where $A_{s,i} = \{s_Q \mid P_i \in Q \text{ for } Q \in \mathcal{Q}\}$, for $i \in \{1, \ldots, n\}$.
- 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:

$$\sum_{Q \in \mathcal{Q}} s_Q = \sum_{j=1}^{k} s_j = s,$$

- 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, players in Q receive s_Q and $s_{Q'}$ for all $Q' \in \mathcal{Q} \setminus Q$, and thus 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 [6] 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 4.2.

4.1.2.2 Operations on the shares.

Some methods are needed for players to be able to do some operations on shared secrets. These operations 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. The elements s_Q are referred to as *shares* of the secret s, for all $Q \in \mathcal{Q}$. Each player knows only a certain number of such shares. A *public value* $c \in \mathbb{F}_p$ is not written inside brackets and it is known by all players. When writing $[[s]] + c$ we are referring to a shared secret $[[x]]$ such that when opening $[[x]]$, 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.

(i) Having $[[s]]$ and $[[t]]$, to compute $[[s + t]]$.

(ii) Having $[[s]]$ and a public $c \in \mathbb{F}_p$, to compute $[[s]] + c$.

(iii) Having $[[s]]$ and a public $c \in \mathbb{F}_p$, to compute $[[s]] \cdot c$.

(iv) 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 latter 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 computing $[[s]] + c$

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

(i) A set $Q \in \mathcal{Q}$ is agreed.

(ii) The players in Q perform the operation $s_Q + c$. The other players do not perform any addition.

(iii) This way, we can redefine the shares with a new share $[[x]] = (A_{x,1}, \ldots, A_{x,n})$, 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 $[[x]]$[2] players can compute:

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

4.1.2.3 Multiplication protocols.

The last operation to study is multiplication of $[[s]]$ and $[[t]]$. Unfortunately and unlike the previous operations, there is no known manner to make multiplication possible without involving communication between different players.

Recall that (Γ, Δ) is assumed to be Q_2. From Definition 4.13 this means that there do not exist $U_1, U_2 \in \Delta$ such that

$$P = U_1 \cup U_2.$$

[2]Refer to protocol OPEN.Reveal() in Section 4.2 to see how to open $[[x]]$.

Let us express this condition in terms of the sets in $\mathcal{Q}$. 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 [3], Maurer introduced a protocol for multiplying two shared secrets. The protocol, detailed in the following lines, is used in the offline phase of the Shamir Protocol.

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.

(i) Split the set $\mathcal{T} = \{(i,j) \mid 1 \leq i, j \leq k\}$ into:

- $\mathcal{T}_1 = \{(i,j) \mid P_1 \in Q_i \cap Q_j\}$
- $\mathcal{T}_2 = \{(i,j) \mid P_2 \in Q_i \cap Q_j\}$
- $\vdots$
- $\mathcal{T}_n = \{(i,j) \mid P_n \in Q_i \cap Q_j\}$

(ii) Each player P_m with $m \in \{1, \ldots, n\}$ computes

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

Observe that the previous step assigned the shares to players, in order to avoid repeated shares among players when calculating the above addition.

(iii) 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 $[[v_1]], \ldots, [[v_n]]$.

(iv) 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 calling protocol OPEN.Reveal($[[z]], j$) for $j \in \{0, \ldots, n\}$ in Section 4.2, 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. That is why this protocol is useless for an online phase, which is supposed to be fast. LSSS-based SMPC protocols tend to perform all heavy computations in the offline phase, in order to have only light computations in the online phase. For this purpose, the so called Beaver triples are essential, as they introduce a faster multiplicative method once the protocol above is done in the offline phase.

4.1.2.4 Beaver triples.

Let us give a glance of Beaver triples and their properties (mainly, the reason why the product holds when using them with other shared values). In the literature the definition of Beaver triples is the following.

Definition 4.18. 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:

(i) Compute:

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

(ii) Call the functions OPEN.Reveal($[[\alpha]]$, 0) and OPEN.Reveal($[[\gamma]]$, 0). Then all players have α and γ.

(iii) 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:

$$[[z]] = [[a \cdot b + (x - a) \cdot b + (y - b) \cdot a + (x - a) \cdot (y - b)]]. \tag{4.1}$$

This formula comes from the fact that:

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

Simplifying (4.1) 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.

4.2 Shamir Protocol overview.

In the Shamir Protocol, as in most common LSSS-based SMPC protocols, there are two main phases: the offline phase and the online phase. At the beginning of the protocol, players have some private inputs, and some computations to do over those inputs. Those computations are given in the form of a circuit. In each phase the players do the following:

- Offline phase: In this phase, the players prepare all the elements needed to evaluate the circuit. This preparation includes the creation of multiplication triples, square triples, etc.
- Online phase: In this phase, the players call the LSSS.PRSS protocol to share their private inputs, and then they are ready to evaluate the circuit. Once they have performed all the computations required in the circuit, they open the resulting secrets in order to obtain the output.

The aim of this section is to describe with the maximum detail both of these phases. The offline phase is studied in Section 4.2.1 and the online phase in Section 4.2.2. But before moving forward, we will introduce two auxiliary protocols that will be used in those 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 $v \in \mathbb{F}_p$. We establish that 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 such 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 from now on (see [6] for the discussion on the existence of partitions of $\mathcal{Q}$ with non-empty sets).

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):
  - (i) P_j sends m to all players over authenticated channels.
  - (ii) 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 be able to compute 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 $\{\mathcal{Q}_i\}_{i=1}^n$ is assumed to satisfy that all $\mathcal{Q}_i$ must be non-empty, we know that at least there exists some $Q \in \mathcal{Q}_i$:
    - (i) 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 $s = \sum_{Q \in \mathcal{Q}} s_Q$.
    - (ii) Every player calls HASH.Update(s_Q) $\forall Q \in \mathcal{Q}$.
    - (iii) The players call OPEN.CompareView().
  - If $j \neq 0$ then the secret will be opened only to P_j:
    - (i) 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.

(ii) If any two shares from 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():
  (i) Each $P_i \in P$ executes HASH.Finalise() and gets an output h_i.
  (ii) Each $P_i \in P$ sends h_i to the all other players.
  (iii) Each player compares all hashes $\{h_i\}$ received and:
    – If all hashes are equal then the player does not abort and runs HASH.Init().
    – Otherwise the player aborts and runs HASH.Init().

4.2.1 The offline phase.

As seen in Section 4.1.2.4, 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 LSSS-based 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 [8] 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 [8]. 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 its 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 $\vec{\beta}_1, \dots, \vec{\beta}_{n_T} \in \vec{\beta}$ to denote the buckets inside of $\vec{\beta}$, so $\vec{\beta} = \{\vec{\beta}_1, \dots, \vec{\beta}_{n_T}\}$.

- 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 $j \in \{1, \dots, K\}$ and $i \in \{1, \dots, n_T\}$.

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 [8] it is shown that $M = (n_T + SL)(K - 1) + n_T$.

(i) 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 $([[a_j]], [[b_j]])$.

(ii) 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 $([[a_j]], [[b_j]], [[c_j]])$.

(iii) 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 $n_T + SL$.

(iv) 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 $n_T/L + S$.

(v) 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 $\vec{D_{k,i}}$.

(vi) For $k \in \{2, \ldots, K\}$ players jointly and securely shuffle each $\vec{D_{k,i}}$ in $\vec{D_k}$.

(vii) 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 $\vec{D_{k,i}}$.

(viii) 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, by Section 4.1.2.2)
- $\tau = r \cdot [[a_1^i]] - [[a_j^i]]$ (can be computed locally, by Section 4.1.2.2)

(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:

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

4.2.2 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 share their inputs performing an LSSS.PRSS()-based secret sharing protocol, 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.

It is important to say that the Shamir Protocol is based on the online protocol in [9]. However, that document is based on *monotone span programs* (MSP's), which are special matrixes that can be used as a model of computation. These matrixes are of special interest, on the one hand, because it is well known that, given a monotone access structure (Γ, Δ), there exists an MSP computing it (see [10], [11], [12]). On the other hand, they have a close relationship with secret sharing schemes, as shown in [9]. Considering these two facts, it seems natural to use them for the SMPC protocols. Nevertheless, the online protocol in [9] can be introduced without the need of MSP's, following the notation introduced in this paper work so far. As a result, we obtain a more intuitive online phase. If the reader is interested in MSP's, we refer to [9].

Protocol ONLINE

- Init(n_T):
  - (i) Every player calls HASH.Init().
  - (ii) 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.
  - (i) The players jointly call LSSS.PRSS() to obtain $[[r]]$, where r is a random value. Recall that LSSS.PRSS() was the secret sharing protocol.
  - (ii) The players call OPEN.Reveal($[[r]], j$) to open $[[r]]$ for P_j. Now P_j knows r.
  - (iii) P_j calculates $\epsilon = x - r$ and calls OPEN.Broadcast(j, ϵ) to send ϵ to all players.

(iv) 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.
  (i) Take one unused triple from the ones generated in OFFLINE protocol. Suppose that the triple taken is $([[a]], [[b]], [[c]])$, where $c = a \cdot b$.
  (ii) 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 $[[s]]$.
  (i) The players run OPEN.CompareView(). If the protocol results in abort, the players abort the output protocol. If not, continue.
  (ii) 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.

Chapter 5

The final payments.

Section 3.3 described a protocol to pack players into bins before performing the SMPC in each bin. In this section we assume the SMPC is finished. At this stage, each participating player will have obtained one between two possible outputs: **SUCCESS** or **ABORT**. In fact, the SMPC ends successfully if and only if all players obtained **SUCCESS**.

Intuitively, the payment process will work as follows: if most of players 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

From now on, for the sake of simplicity, we will assume that there is only one bin, since the payment process is equal in each bin. Let the bin have n players (the scientist included). When performing the SMPC 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. In other words, we assume that all players will automatically generate and send a transaction to the Smart Contract, reporting the result they obtained from the SMPC. This transaction is assumed to happen during the SMPC protocol, and thus only those 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 2.2.
  - Reputations are increased. Recall from Section 2.2 that a player with reputation R will gamble part of their reputation $r < R$. In fact, from Section 2.2, 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.
- Exactly $n/2$ players report **SUCCESS** and the other half reports **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**.

5.1 The actual payments.

This section gives the details for data owners to know an approximation of the payments they will receive if the SMPC ends up succesfully. Thus, consider a data scientist has recently published an SMPC request. We assume a data owner who wants to sign up for that request, which has MODL $\mathcal{L}$ and total payment $\mathcal{P}$. We let the size of the data owner's dataset be w. From Section 3.2, data owners know that they will receive

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

tokens in case the SMPC succeeds. Recall from Section 3.2 that $\rho = \frac{\mathcal{P}}{\mathcal{L}}w$. Therefore, data owners have an approximation of the tokens they might receive before signing up for the request.

This section gives the details to establish the lower and upper boundaries for $\hat{\rho}$. The discrepancy between ρ and $\hat{\rho}$ is due to the fact that $\mathcal{P}$ is fixed before data owners sign up for the request. Hence, the actual overall dataset length might differ from $\mathcal{L}$. Indeed, suppose a scientist threw a request with MODL $\mathcal{L}$ and that the data providers who signed up had dataset lengths $W = \{w_1, \ldots, w_n\}$. 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 (3.1), but with

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

for all $P_i \in P$ where $i \in \{1, \ldots, n\}$, and where $\sum_{j=1}^{n} w_j \geq \mathcal{L}$.

We next describe how good of an approximation is

$$\rho_i = \frac{\mathcal{P}}{\mathcal{L}} \cdot w_i$$

of the true payment $\hat{\rho}_i$ for P_i. On the one hand, we can easily observe that the largest possible $\hat{\rho}_i$ is attained when $\sum_{j=1}^{n} w_j = \mathcal{L}$. In this situation we have:

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

On the other hand, recall from the Section 3.2 that we required each w_i to be $w_i < \mathcal{L}$, and that the sign up process finishes once there are enough players $P_1, \ldots, P_n$ such that

$$\sum_{i=1}^{n} w_i \geq \mathcal{L}.$$

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 SMPC request 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:

$$\hat{\rho}_i = \frac{\mathcal{P}}{\sum_{j=1}^{n} w_j} w_i = \frac{\mathcal{P}}{2\mathcal{L} - 2} w_i \approx \frac{1}{2} \rho_i.$$

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

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

Chapter 6

Conclusions.

6.1 A brief summary of the protocol.

Summarizing all the processes studied in this paper, we obtain the following protocol. We assume at least $n/2$ players behave correctly during the SMPC.

(i) A data scientist makes 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 gamble to be able to subscribe.
- The total reputation the scientist will gamble.

(ii) Data owners receive the previous request and if it fits to them they sign up to it by specifying:

- The reputation they will gamble in this SMPC (must be greater than the minimum required by the scientist).
- The data length they have to contribute in the SMPC.

(iii) When all the owners' data lengths surpass the MODL, the subscribing time ends and the data scientist starts the grouping protocol. At this stage, players will know exactly the amount of tokens $\hat{\rho}$ they will be paid for the job, and the bin they are assigned to for performing the computations.

(iv) During the SMPC, an automatically generated transaction is sent to the Smart Contract (one per player). This transaction reports to the blockchain if the SMPC ended with **SUCCESS** or with **ABORT**.

(v) 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.

Bibliography

[1] Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2009).

[2] Yao, A.C.C.: How to generate and exchange secrets (extended abstract). In 27th FOCS. pp. 162–167. IEEE Computer Society Press (1986).

[3] Maurer, U.: Secure Multi-Party Computation Made Simple. Discrete Applied Mathematics 154(2), 380-381 (2006).

[4] Beaver, D., Wool, A.: Quorum-based secure multi-party computation. In Nyberg, K. (ed.) EUROCRYPT'98. LNCS, vol. 1403, pp. 375–390. Springer, Heidelberg (1998).

[5] Damgard, I., Pastro, V., Smart, N. P. and Zakarias, S.: MultiParty Computation from Somewhat Homomorphic Encryption. In Safavi-Naini, R., Canetti, R.: CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012).

[6] Keller, M., Rotaru, D., Smart, N. P., and Wood, T. Reducing communication channels in MPC. In Catalano, D. and De Prisco, R.: Security and Cryptography for Networks. 11th International Conference, 2018. Proceedings, vol. 11035 of Lecture Notes in Computer Science, pp. 181–199. Springer (2018).

[7] Smart, N.P. et al.: SCALE-MAMBA v1.12: Documentation (2021).

[8] Araki T., Barak A., Furukawa J., Lichter T., Lindell Y., Nof A., Ohara K., Watzman A., Weinstein O.: Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In 2017 IEEE Symposium on Security and Privacy, pp. 843–862 (2017).

[9] Smart N.P., Wood, T.: Error detection in monotone span programs with application to communication-efficient multi-party computation. In Matsui M.: RSA Conference 2019. The Cryptographers'

Track. Proceedings, vol. 11405 of Lecture Notes in Computer Science, pp. 210–229. Springer (2019).

[10] van Dijk, M.: Secret Key Sharing and Secret Key Generation. Ph.D. thesis, Eind-hoven University of Technology (1997)

[11] Ito, M., Saito, A., Nishizeki, T.: Secret sharing schemes realizing general access structure. In Proc. IEEE Global Telecommunication Conf. pp. 99–102 (1987)

[12] Karchmer, M., Wigderson, A.: On span programs. In Proceedings of Structures in Complexity Theory. pp. 102–111 (1993)

[13] Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 169–188. Springer, Heidelberg (2011).

[14] Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 1–18. Springer, Heidelberg (2013).

[15] Keller, M., Orsini, E., Scholl, P.: MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 16. pp. 830–842. ACM Press (2016).

[16] https://github.com/rdragos/awesome-mpc

[17] Assmann, S. F.: Problems in Discrete Applied Mathematics, Ph.D. Thesis, Mathematics Department, MIT (1983).

[18] Assmann, S. F., Johnson, D. S., Kleitman, D. J., Leung, J. Y.-T: On a dual version of the one-dimensional bin packing problem. In Journal of Algorithms 5, pp. 502-525 (1984).

[19] Csirik, J., Frenk, J. B. G., Labbé, M., Zhang, S.: Two simple algorithms for bin covering. In Acta Cybernetica 14, pp. 13-25 (1999).

[20] Csirik, J., Johnson, D. S., Kenyon, C.: Better approximation algorithms for bin covering. In Proceedings of the 12th annual ACM-SIAM symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, pp. 557-566 (2001).

[21] Jansen, K., Solis-Oba, R.: An Asymptotic Fully Polynomial Time Approximation Scheme for Bin Covering. In Proceedings of the 13th

International Symposium on Algorithms and Computation, Springer-Verlag pp. 175-186 (2002).

[22] Koch, K., Krenn, S., Pellegrino, D., Ramacher, S.: Privacy-preserving Analytics for Data Markets using MPC. In IFIP Summer School on Privacy and Identity Management 15, Springer (2020).

[23] Eyal, I. et al.: White-City: A Framework For Massive MPC with Partial Synchrony and Partially Authenticated Channels.

[24] Zhang, D., Su, A., Xu, F., Chen, J.: ARPA Whitepaper.

[25] https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts/token/ERC20

[26] https://ipfs.io/

[27] https://github.com/eventeum/eventeum

[28] https://github.com/julenbernabe/SMPC-BCK

Appendix A

Implementation

A.1 The Smart Contract.

The Smart Contract implemented is introduced in the following lines. This contract is written in Solidity language, and imports other contract implementing a token. For details of such an implementation, refer to [25].

```
    // SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;


import "./token/Token.sol";

contract subscribe {

    /* --------------- OTHER CONTRACT IMPORTATION --------------- */


    /* We first of all initialize the SocialToken contract's
        address in this contract
    since we will need to do payments in the Universe network.*/

    Token private token;
    address private tokenaddress =
        0x8D5a5965338AaE0FbC02B16D1720c29f92CD7c94;


    // -------------------------CONSTRUCTOR-------------------------


    // In constructor we initialize the contract state and the
        current operation to the default values.
```

```
    constructor() {
        last = 0;
        token = Token(tokenaddress);
    }

    // -------------------------STRUCTS-------------------------


    // First struct. The general one. Records all data about the
        program that will be computed.

    struct ProgData {
        uint n_players;   // The number of players in the SM network
        string CA;        // The CA certificate, for players to sign
            their CRTs with it
        uint reputation;
        uint prime;       // The prime we will use for the operations
        uint subscribed_players;
        uint payment;
        uint tokensPerItem;
        uint MODL;
        uint current_data;
        uint ready_players;
        uint finished_players;
        uint succeeded;
        uint scientist_vote;
        uint aborted;
        mapping (address => uint) elected;
        mapping (uint => PlayerData) Players;   // This mapping will
            connect an integer (0-n_players) to each player's data:
            IP, CRT and CN
        FilesLoc Files;   // In Files we record the locations
            (hashes) of the program files on IPFS.
        StateType ContractState;                // Here we update
            the state of the contract for each thrown program.
    }

    // Second struct. Inside ProgData. We record the data of each
        player.

    struct PlayerData {
        address account;  // We record the account for the future
            rewards that will be given to the players
        string IP;        // The IP, CRT and CN of each player need
            to be recorded for the Setup of SM to be able to prove
            those CRTs
        string CRT;       // and write them on NetworkData.txt
```

```
    string CN;
    uint data_length;
    uint reputation;
}

// Third struct. Inside ProgData too. We record the locations
    of the files needed to run the mpc program in IPFS.

struct FilesLoc {
    string BC;        // Hash of the folder containing the .bc
        and .sch files
    string SCH;
    string PREP;      // A program to preprocess the Excel table
        where the private data is. This program outputs a txt
                      // where the data of the Excel file will
                          be readable for the BC
    string INFO;      // A file where some information is given
        about the program that will be computed and, of course,
        about
                      // the kind of data that will be needed
                          (type, units, how to write the
                          Excel...)
}

// -------------------------EVENTS-------------------------

// First event. This will log in the blockchain that an
    operation has been thrown. This way, all players will be
    able to know that
// an operation is about to happen and that it's time to sign
    up to it.

event newRequest(
    uint i,
    string info
);

event updatedRequest(
    uint i,
    string info
);

event operationThrown(
    uint i
);

event operation_finished(
    uint i
);
```

```
// This event will tell the scientist if it's time to start the
    SM part.

event playersFilled(
    uint i
);

// This event tells players that everyone has done the first
    Setup in SM. This way, they know that they can run the
    program securely.

event network_ready(
    uint i
);

// ----------------------OTHER VARIABLES----------------------

// This mapping keeps in its keys all the ProgData structs and
    their information.

mapping (uint => ProgData) Prog;

// These mappings record the reputations of players in the
    network.

mapping (address => uint) registered;
mapping (address => uint) reputations;

uint public last;                // The last item that has a
    struct in Prog.

// There will be 6 states in this contract. The contract will
    give access to functions depending on its current state.

enum StateType {ThrowOperation, SubscribingTime,
    OperationPending, OperationThrown, NetworkReady, Finished}


// ------------------- CALLDATA FUNCTIONS -------------------

// The following function will return the actual contract state
    of a program for a user who wants to know.

function return_state(uint i) external view returns (StateType,
    string memory, string memory, uint) {
    ProgData storage NewProg = Prog[i];
    string memory text1;
    string memory text2;
```

```
        uint number;
        if (NewProg.ContractState == StateType.SubscribingTime) {
            text1 = "We are at subscribing time";
            text2 = "Subscribed players:";
            number = NewProg.subscribed_players;
        }
        else if (NewProg.ContractState ==
            StateType.OperationPending) {
            text1 = "We are waiting for the data scientist to throw
                the operation";
            text2 = "Subscribed players:";
            number = NewProg.subscribed_players;
        }
        else if (NewProg.ContractState == StateType.OperationThrown)
            {
            text1 = "We are waiting for all players to tell they are
                ready.";
            text2 = "Ready players at this moment";
            number = NewProg.ready_players;
        }
        else if (NewProg.ContractState == StateType.NetworkReady) {
            text1 = "The network is ready to start the SCALE-MAMBA
                computations.";
            text2 = "";
            number = 0;
        }
        return (NewProg.ContractState, text1, text2, number);
    }

    // This function returns Player[n]'s data (the IP, CRT and CN,
        in that order). This function cannot be called until all
        players
    // required are subscribed, and only addresses who singed up to
        the operation can call it.

    function return_PlayerData(uint n, uint i) external view
        returns (uint, string memory, string memory, string memory,
        uint) {
        ProgData storage NewProg = Prog[i];
        require (NewProg.ContractState == StateType.OperationThrown,
            "You cannot execute this function at this moment.");
        require(NewProg.elected[msg.sender] == 1, "You are not
            allowed to obtain this information.");
        require(NewProg.elected[NewProg.Players[n].account] != 0,
            "This player is not elected, you cannot obtain the
            information.");
        uint my_n = 0;
        if (msg.sender == NewProg.Players[n].account) {
            my_n = 1;
```

```
        }
        return (n, NewProg.Players[n].IP, NewProg.Players[n].CRT,
            NewProg.Players[n].CN, my_n);
    }

    function return_subscribed_n(uint i) external view returns
        (uint) {
        ProgData storage NewProg = Prog[i];
        require(NewProg.ContractState == StateType.OperationPending,
            "You cannot execute this function at this moment.");
        require(msg.sender == NewProg.Players[0].account, "You are
            not allowed to obtain this information.");
        return NewProg.subscribed_players;
    }

    function return_PlayerData_private(uint n, uint i) external
        view returns (address, uint) {
        ProgData storage NewProg = Prog[i];
        require(msg.sender == NewProg.Players[0].account, "You
            cannot call this function.");
        require (NewProg.ContractState ==
            StateType.OperationPending, "You cannot execute this
            function at this moment.");
        return (NewProg.Players[n].account,
            NewProg.Players[n].data_length);
    }

    function _elected(uint i) external view returns (uint) {
        ProgData storage NewProg = Prog[i];
        require(NewProg.ContractState == StateType.OperationThrown,
            "You cannot execute this function at this moment.");
        return NewProg.elected[msg.sender];
    }

    // This function returns the prime number that will be used in
        the program. Again, it can only be accesed when all players
        are
    // signed up and they are the only ones that have access to it.

    function return_prime(uint i) external view returns (uint) {
        ProgData storage NewProg = Prog[i];
        require (NewProg.ContractState == StateType.OperationThrown
            || NewProg.ContractState == StateType.Finished, "You
            cannot execute this function at this moment.");
        require(NewProg.elected[msg.sender] == 1, "You are not
            allowed to obtain this information.");
        return NewProg.prime;
    }
```

```
// This function returns the required number of players. This
    function is needed to perform the Setup process properly.
    However,
// it could be of interest to know the required number of
    players whenever a player wants (it's not relevant data).

function return_n_players(uint i) external view returns (uint) {
    ProgData storage NewProg = Prog[i];
    require(NewProg.ContractState == StateType.OperationThrown,
        "You cannot obtain this information at this moment.");
    require(NewProg.elected[msg.sender] == 1, "You are not
        allowed to obtain this information.");
    return NewProg.n_players;
}

// This function returns the location of the documentation file
    about the program. We require, evidently, the program to be
    thrown,
// but this function can be called whenever a user wants. This
    is because this file will tell a player what kind of data is
    needed
// for this program, and how must it be writen in the Excel for
    the BC to be readable.

function return_info(uint i) external view returns (string
    memory) {
    ProgData storage NewProg = Prog[i];
    require (NewProg.ContractState != StateType.ThrowOperation,
        "The file you are searching for is not uploaded yet. If
        you want, you can throw an operation to compute.");
    return NewProg.Files.INFO;
}

// This function returns the locations of the program files
    needed to run the Player. It cannot be called until all
    players are subscribed.

function return_locs(uint i) external view returns (string
    memory, string memory, string memory) {
    ProgData storage NewProg = Prog[i];
    require (NewProg.ContractState == StateType.OperationThrown,
        "You cannot obtain this information yet.");
    require(NewProg.elected[msg.sender] == 1, "You are not
        allowed to obtain this information.");
    return (NewProg.Files.BC, NewProg.Files.SCH,
        NewProg.Files.PREP);
}

// This function returns the CA certificate. Again, this
```

```
    function can be called whenever a user wants, since (I
    think) it could be
// needed before the Setup process.

function return_CA(uint i) external view returns (string
    memory) {
    ProgData storage NewProg = Prog[i];
    require(NewProg.ContractState == StateType.SubscribingTime
        || NewProg.ContractState == StateType.OperationThrown,
        "You cannot obtain this information at this moment.");
    return NewProg.CA;
}

// This function outputs the upper and lower bounds of payments
    for participating in the SMPC of the program.

function return_payment(uint i) external view returns (uint,
    uint) {
    ProgData storage NewProg = Prog[i];
    return (NewProg.payment, NewProg.MODL);
}

// This function returns the reputation of a user.

function return_reputation() external view returns (uint) {
    return reputations[msg.sender];
}

// This function returns the required reputation of a request.

function return_required_reputation(uint i) external view
    returns (uint) {
    ProgData storage NewProg = Prog[i];
    return NewProg.reputation;
}

// This function returns the required reputation of a request.

function return_MODL(uint i) external view returns (uint) {
    ProgData storage NewProg = Prog[i];
    return NewProg.MODL;
}

function return_registered() external view returns (uint) {
    return registered[msg.sender];
}

//This function returns the amount of tokens a user has.
```

```
function balance() external view returns (uint) {
    return token.balanceOf(msg.sender);
}


// --------------------AUXILIARY FUNCTIONS--------------------


// This function returns a 0 if an account has already
    subscribed to the operation, and a 1 if it does not.

function account_repeated (uint j) public view returns (uint) {
    ProgData storage NewProg = Prog[j];
    uint result = 1;
    for (uint i = 0; i < NewProg.subscribed_players; i++) {
        if (NewProg.Players[i].account == msg.sender) {
            result = 0;
            break;
        }
    }
    return result;
}

function withdraw(uint amount) external payable {
    token.transfer(msg.sender, amount);
}

// ----------------------MAIN FUNCTIONS----------------------


// The following function updates the number of players that
    have picked up the SM networking information so far. When
    the number
// of players reaches the required number of players, an event
    is emitted, for the leader to know that the operation can be
    run.

function Im_ready(uint i) external {
    ProgData storage NewProg = Prog[i];
    require(NewProg.ContractState == StateType.OperationThrown,
        "The contract cannot proccess your claim in its current
        state.");
    require(NewProg.elected[msg.sender] == 1, "You are not
        allowed to tell you're ready.");
    NewProg.elected[msg.sender] = 2;
    NewProg.ready_players += 1;
    if (NewProg.ready_players == NewProg.n_players) {
        emit network_ready(i);
        NewProg.ContractState = StateType.NetworkReady;
```

```
    }
}

function finished(uint i, uint success) external {
    ProgData storage NewProg = Prog[i];
    require(NewProg.ContractState == StateType.NetworkReady,
        "The contract cannot proccess your claim in its current
        state.");
    require(NewProg.elected[msg.sender] == 2, "You are not
        allowed to tell you've finished.");
    if (msg.sender == NewProg.Players[0].account) {
        NewProg.scientist_vote = success;
    }
    NewProg.elected[msg.sender] = 3;
    NewProg.finished_players += 1;
    if (success == 0) {
        NewProg.aborted += 1;
    } else {
        NewProg.succeeded += 1;
    }
    if (NewProg.finished_players == NewProg.n_players) {
        emit operation_finished(i);
        if (NewProg.succeeded > NewProg.aborted) {
            pay_to_players(i);
        } else if (NewProg.succeeded == NewProg.aborted) {
            if (NewProg.scientist_vote == 1) {
                pay_to_players(i);
            }
        }
        NewProg.ContractState = StateType.Finished;
    }
}

// All players, when registering in the marketplace, start with
    an initial reputation of 50 points. This reputation may
// increase or decrease depending on their success when
    participating in SMPC requests.

function register() external {
    require(registered[msg.sender] == 0, "You already registered
        in the marketplace.");
    registered[msg.sender] = 1;
    reputations[msg.sender] = 50;
}

// The signing up function is responsible for taking all the
    necessary information about the players that will form the
    SCALE-MAMBA network.
// Inside it, we update the ProgData struct with all the new
```

```
        information about the player. When we reach the required
        number of players,
    // the players_filled event is logged into the blockchain,
        telling the leader that it is time to start the operation.

    function signup(uint my_data_length, string calldata myIP,
        string calldata myCRT, string calldata myCN, uint i, uint
        _reputation) external {
        require(reputations[msg.sender] >= _reputation, "You gambled
            more reputation than the one you have.");
        ProgData storage NewProg = Prog[i];
        require (NewProg.ContractState == StateType.SubscribingTime,
            "The contract cannot proccess your claim in its current
            state.");
        require (account_repeated(i) == 1, "You already have signed
            up for this operation.");
        require (_reputation >= NewProg.reputation, "Your reputation
            is not enough to participate in this SMPC.");
        require (my_data_length < NewProg.MODL, "Your data length
            must be less than the MODL.");
        reputations[msg.sender] -= _reputation;
        NewProg.Players[NewProg.subscribed_players] = PlayerData(
            {
                account: msg.sender,
                IP: myIP,
                CRT: myCRT,
                CN: myCN,
                data_length: my_data_length,
                reputation: _reputation
            }
        );
        NewProg.subscribed_players += 1;
        NewProg.current_data += my_data_length;
        if (NewProg.current_data >= NewProg.MODL) {
            NewProg.ContractState = StateType.OperationPending;
            emit playersFilled(i);
        }
    }

    function update_request(uint i, string[4] memory str_items,
        uint my_data_length, uint _MODL, uint min_reputation, uint
        tokens) external {
        // str_items = [CA, IP, CRT, CN]
        require(reputations[msg.sender] >= min_reputation, "You
            required more reputation than the one you have.");
        require(token.balanceOf(msg.sender) >= tokens, "You promised
            more tokens than the ones you have.");
        require (i <= last, "There is no previous operation to
            update at this index.");
```

```
        ProgData storage NewProg = Prog[i];
        require (NewProg.ContractState == StateType.Finished, "This
            program has not finished yet. You cannot update it.");
        for (uint j = 0; j < NewProg.subscribed_players; j++) {
            NewProg.elected[NewProg.Players[j].account] = 0;
        }
        NewProg.CA = str_items[0];
        NewProg.reputation = min_reputation;
        reputations[msg.sender] -= min_reputation;
        NewProg.ContractState = StateType.SubscribingTime;
        NewProg.ready_players = 0;
        NewProg.finished_players = 0;
        NewProg.aborted = 0;
        NewProg.succeeded = 0;
        NewProg.scientist_vote = 0;
        NewProg.subscribed_players = 1;
        NewProg.MODL = _MODL;
        NewProg.payment = tokens;
        NewProg.current_data = my_data_length;
        NewProg.Players[0] = PlayerData(
            {
                account: msg.sender,
                IP: str_items[1],
                CRT: str_items[2],
                CN: str_items[3],
                data_length: my_data_length,
                reputation: min_reputation
            }
        );

        // Before calling this function we need to call the
            approve(tokenaddress, amount) from Token.sol

        token.transferFrom(msg.sender, address(this), tokens);
        emit updatedRequest(i, NewProg.Files.INFO);
    }

    // This function is the one that is called when throwing an
        operation. It updates the struct of the contract and tells
        the other
    // users that a new operation is thrown.

    function new_request(string[6] memory str_items, uint
        my_data_length, uint _MODL, uint min_reputation, uint
        tokens) external {
        // str_items = [CA, IP, CRT, CN, PREPROCESSING, INFORMATION]
        require(reputations[msg.sender] >= min_reputation, "You
            required more reputation than the one you have.");
        require(token.balanceOf(msg.sender) >= tokens, "You promised
```

```
            more tokens than the ones you have.");
        last = last + 1;
        ProgData storage NewProg = Prog[last];
        NewProg.CA = str_items[0];
        NewProg.reputation = min_reputation;
        reputations[msg.sender] -= min_reputation;
        NewProg.ContractState = StateType.SubscribingTime;
        NewProg.subscribed_players = 1;
        NewProg.MODL = _MODL;
        NewProg.payment = tokens;
        NewProg.current_data = my_data_length;
        NewProg.Players[0] = PlayerData(
            {
                account: msg.sender,
                IP: str_items[1],
                CRT: str_items[2],
                CN: str_items[3],
                data_length: my_data_length,
                reputation: min_reputation
            }
        );
        NewProg.Files.PREP = str_items[4];
        NewProg.Files.INFO = str_items[5];

        // Before calling this function we need to call the
            approve(tokenaddress, amount) from Token.sol

        token.transferFrom(msg.sender, address(this), tokens);
        emit newRequest(last, str_items[5]);
    }

    function throw_operation(uint prime, address[] calldata
        elected_players, string calldata BC_hash, string calldata
        SCH_hash, uint i, uint actual_payment) external {
        ProgData storage NewProg = Prog[i];
        require(NewProg.ContractState == StateType.OperationPending,
            "You cannot call this function at this moment.");
        require(msg.sender == NewProg.Players[0].account, "You are
            not who made the request for this operation.");
        NewProg.prime = prime;
        NewProg.tokensPerItem = actual_payment;
        NewProg.n_players = elected_players.length;
        for (uint j = 0; j < NewProg.n_players; j++) {
            NewProg.elected[elected_players[j]] = 1;
        }
        NewProg.elected[msg.sender] = 1;
        NewProg.n_players += 1;
        NewProg.Files.BC = BC_hash;
        NewProg.Files.SCH = SCH_hash;
```

```
        NewProg.ContractState = StateType.OperationThrown;
        emit operationThrown(i);
    }

    function pay_to_players(uint i) public payable {
        ProgData storage NewProg = Prog[i];
        uint j;
        reputations[NewProg.Players[0].account] += 2 *
            NewProg.reputation;
        for (j = 1; j < NewProg.n_players; j++) {
            if (NewProg.elected[NewProg.Players[j].account] != 0) {
                reputations[NewProg.Players[j].account] += 2 *
                    NewProg.Players[j].reputation;
                uint amount = NewProg.tokensPerItem *
                    NewProg.Players[j].data_length;
                token.transfer(NewProg.Players[j].account, amount);
            }
        }
    }
}
```

A.2 The Python client.

The Python client gives the users the necessary tools to interact with the above Smart Contract. Furthermore, it integrates other tools, such as IPFS [26], Eventeum [27]... For a detailed study of this client, refer to the GitHub repository in [28].