CS-523 SMCompiler Report

Aybars Yazıcı, Can Kırımca

***Abstract*—In this report, we report on our implementation and results of our evaluation for the first project of CS-523 conducted at EPFL during the Spring 2023 session.**

I. Introduction

The goal of this project to impelement a Secure Multi Party(shortened to SMC in this paper from now on) framework, which allows multiple parties to compute a function on their private data without revealing any information about their data to each other. The framework should be able to handle addition, substraction and multiplication operations on Scalars and Secrets. More information about this is covered in III. After implementing the framework we performed performance evaluation by varying both the party size and Secret amount, to understand the amount of bytes exchanged and the computation time for each operation, which is explained in IV The final part of the paper V covers a possible use case of this framework,it's implementation details and the unit tests that we have performed to ensure the correctness of our implementation.

II. Threat model

The critical part that should be clarified first about this implementation is the assumption. The framework assumes honest but curious clients, as any deviation from the given algorithm would break the framework. On top of this all parties are required to cooperate to be able to retrieve the final results. I.e. if there are N parties, there will be N shares computed, and N shares are required to construct the final result. The details are further explained in III.
Another point that should be clarified is the communication between clients. In our implementation there is no pairwise connection between the clients, instead there is a central server which the clients use to send private and public messages. Here we reveal another assumption, as the server could in reality pool all the secrets sent and reconstruct the secrets. We assume the server is trusted and is not malicious.
The last assumption we have is required to implement the multiplication operation. To be able to implement this operation, we require beaver triplets to be generated. We assume that the beaver triplets are generated by a trusted third party, which is not part of the SMC framework,i.e it does not participate in the computations. The details of this assumption is covered in III.

III. Implementation details

Our code was built on top of the skeleton code provided. This section is divided into subsections, which represents the steps taken while implementing the protocol.

A. Constructing the expression tree

The first step was to implement the expression tree, which is the core of the framework. The expression tree is a binary tree, where each node represents an operation. The leaves of the tree are the secrets and scalars. The tree is constructed by parsing the expression given by the user. The expression tree is then processed by the client in a recursive manner, which is explained in III-B. To be able to construct the expression tree, we had to override the operators for the Secret and Scalar classes. The operators are overloaded to return a new node, which represents the operation.
To be able to test the correctness of our expression trees, we have implemented three functions to print the tree in human readable format. First one prints the expression as a string. The second one prints the tree in a unix tree format, where each node is represented by a line. The third one prints the tree in a tree format, with left and right childs. Then we had the test_expression.py file to test if the constructed tree is equal to the expected construction. Now that we had a correct expression tree, we could move on to the next step.

B. Processing the expression

When the protocol runs, each client is given the expression to process, which is represented as a tree as we have previously explained. Before processing the tree each client computes shares of all of their secrets and sends one share to each other client. This is done to ensure that each client has a share of all secrets. After this process each client processes the expression tree. The tree is then recursively processed by the client. The client processes the tree by first processing the left child. Then it processes the right child. After processing the children, the client processes the current node. The processing of the node depends on the type of the node. If the node is a leaf, then it could either be a Secret or a Scalar. If the node is a Secret, then the client gets the his share of this secret from the server and returns, transforming the Secret Node into a Share Node. If the node is a Scalar, then the client returns the scalar(int) value. If the node is an operation, the client returns left_child *operation* right_child. If both children of an operation is a int(Scalar) then the operation is already defined in python. But to be able to do the computations, if any of the children is a Share, the operations and their reverse versions are overloaded in the Share class.
If the operation is a + or - operation, then the client can locally compute the result. If the operation is a * operation, then the client needs to obtain beaver triplets from the server. Note that for each multiplication operation, the client obtains new beaver triplets.
Each evaluation of an operation node returns either a Share or

an int. The operation will return a Share if either left or right child is a Share. If both children are ints, then the operation will return an int. The tree is processed this way from the leafs to the root. The root of the tree is the final result of the expression, which will be a share(assuming there was at least one secret in the expression).
The clients then after obtaining the final results, publishes it as a public message on the server. Then every client retrieves all the public results and reconstructs the final result. If the result obtained is of type int rather than share, then this means that there was no secret in the expression. In this case the client can directly return the result to the user.

IV. Performance evaluation

Report the computation and communication cost of the framework for different circuit and protocol parameters.

V. Application

A. Introduction

For our use case, we imagine the following scenario. Where the clients of our SMC protocol will be the students of 'Evil School'. Where in this school, professors do not announce the statistics for the exams. Instead they just reveal to each student their grade. The students want to be able to know how well they have done in the exam compared to other students, and thus want to learn the mean and standard deviation of the grades. And as the professors don't reveal these, the students want to calculate themselves, but they do not wish the reveal their grades to each other. So they will be using the SMC framework.

B. Adversarial Model

The Adversarial model is as extremely similar to the one in section II. The students are honest but curious. They want to learn the statistics of the grades, but they do not want to reveal their grades to each other. As reconstruction of the secrets require all the shares yet again, if there is any malicious student in the group, then the protocol will fail.
Yet again sending messages/values to each other is done through a server. And Trusted Third Party is used for beaver triplet generation.

C. Averaging

The structure of the use case is as follows: N students have their grades for L amount of classes. Thus in the end each student should obtain L averages and L standard deviations. Firstly each student will generate N many shares for each of their grade and send one to each other. Thus each student will have L shares from each other, which means N*L shares in total. They'll use these shares first to calculate the average for each course. $avg_L = \sum_{i=1}^{N} x_i$ where avg_L is the mean for course L and x_i is the share of the i'th student for the course. Then they'll use these averages to calculate the standard deviation for each course. Note that, to avoid dealing with floating point numbers, we delay the divisions by N until the end of the computation.

D. Standard deviation

Normally standard deviation is calculated as follows: $\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$ But again to not deal with floating point numbers with shares, we'll reserve the square root and division to the last part. So we need to first calculate $\sum_{i=1}^{N}(x_i - \mu)^2$ But because we have not divided the sum by N, we don't currently have μ but rather we have $N\mu$ So with some quick algebra we get the following: $\frac{N^2\sum_{i=1}^{N}(x_i-\mu)^2}{N^2} = \frac{\sum_{i=1}^{N}(N(x_i-\mu))^2}{N^2}$ We again delay the division to the end, so we what each client will calculate is actually: $\sum_{i=1}^{N}(N(x_i - \mu))^2$ Remember that in the previous step they had already calculated $N\mu$ So for each course they'll multiply each share by N substract from mean and square it. They'll sum each of these squares, publish their result then reconstruct the final result. But remember that we have not done any division yet, So before returning the average we divide the $N\mu$ for each course by N, and before returning the standard deviation we divide by N^3 and take the square root.

E. Testing

The proposed system is implemented in use_case.py and tested in test_use_case.py and holds the desired functionality. In test cases we used numpy to calculate the expected averages and standard deviations, and used isclose to check if the results are close enough(within 0.001 margin) to the expected results.