

# 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 implement 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, subtraction 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, its 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.

#### IV. PERFORMANCE EVALUATION

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

#### V. APPLICATION

Detail the use case of SMC and a circuit for this use case. Discuss possible privacy leakage not covered by SMC. Discuss a mitigation if needed.