# CS-523 SMCompiler Report

Rassene M'Sadaa, Luna Ralet, Leila Sidjanski

*Abstract*—**Please report your design, implementation details, findings of the first project in this report.**
**You can add references if necessary [?].**
**Remember to add a paragraph describing the contributions of each team member.**
**THE REPORT SHOULD NOT EXCEED 2 PAGES.**

## I. Introduction

The project goal is to implement a privacy-preserving computation protocol based on Secure Multiparty Computation (SMC). The aim is to allow multiple parties to jointly compute a function over their private inputs without revealing their own inputs or learning the inputs of others.

At the core of the protocol, there is a secret sharing mechanism (`secret_sharing.py`), which splits the (sensitive) data into multiple shares (equal to the number of participants) distributed among the participants. These shares are then exchanged securely between all parties over private communication server. Once every party has received the other's shares, they individually evaluate the function on their local shares using the logic defined in `expression.py`. Finally, the results of these computations are sent over a public channel, allowing all parties to collaboratively reconstruct the final output of the computation. This project is designed to demonstrate the feasibility and performance of secure computations in a simulated multiparty environment.

The overall aim of this protocol is to ensure data privacy during computation, even in the presence of honest-but-curious adversaries.

## II. Authors contributions

Initially, each team member took the time to understand the project and the tasks involved. Leila and Luna began working on the file `secret_sharing.py`, thinking through how each secrets would be split between the participants. They encountered an issue with the choice of modulo and consulted Rassene for his input, which helped resolve the problem. They also implemented the `expression.py` file. Next, Leila and Luna started implementing `smc_party.py`, beginning with the run method. Meanwhile, Rassene focused on implementing `ttp.py` and the Beaver Triplet Protocol (process_mult_op() in `smc_party.py`). Initially, the run() method had a bug where the receiving of the shares was incorrectly iterating over the client's own shares. Rassene noticed the issue and suggested a fix, which Leila and Luna applied under his guidance. Rassene also started implementing process_expression() in `smc_party.py`, while Leila and Luna added support for constant addition. They also suggested adjustments to avoid using the Beaver Triplet Protocol for scalar multiplications. Finally, all three worked together to modify run() so that the number of secrets each client holds could be tracked. Leila and Luna proposed and implemented a custom Application, while Rassene focused on performance testing.

## III. Threat model

We assume that our model is a honest-but-curious adversarial model where all parties follow the protocol correctly but may attemps to deduce additional information from the received output. In this model, all parties follow the protocol as intended, without modifying messages or collaborating with others to compromise privacy. The model uses additive secret sharing over a finite field to securely split private input, and parties exchange share via authenticated and private channels. Additionally, the protocol requires each party to publish their individual share of the computed result over a public channel. For secure multiplication of secret values, the systems uses the Beaver triplets protocol, which are random shared values generated and distributed by a Trusted Third Party (TTP). We assume that the TTP is fully honest and does not conspire with any party or leak sensitive information. Communication is assumed to be secure, and there is no protection against malicious adversaries. Under theses assumptions, the protocol is secure and guarantees that no party can learn anything other than the final output of the computation.

## IV. Implementation Details

Our implementation performs all calculations modulo a Mersenne prime for efficiency and simplicity in modular arithmetic.

To generate shares of a secret, we randomly choose $n-1$ values and compute the final value such that the sum of all shares equals the secret. This ensures correctness while maintaining randomness in the shares.

In the SMC (Secure Multi-Party Computation) party component, each user may own multiple secrets. To determine how many secrets need to be retrieved, we use a recursive function that collects all secret IDs referenced within an expression.

The evaluation of expressions in the SMC party is also done recursively. When the recursion reaches a leaf node that is a scalar, it is converted into a share of that scalar value. To optimize performance, if a multiplication involves a scalar and a share, we skip the use of Beaver triples for that operation.

For performance monitoring, we require access to the number of bytes sent and received during computation. To facilitate this, a 'Performance' class instance is passed to the SMC party via its constructor. This argument is optional (defaults to 'None') and does not interfere with normal test execution when performance tracking is not required.

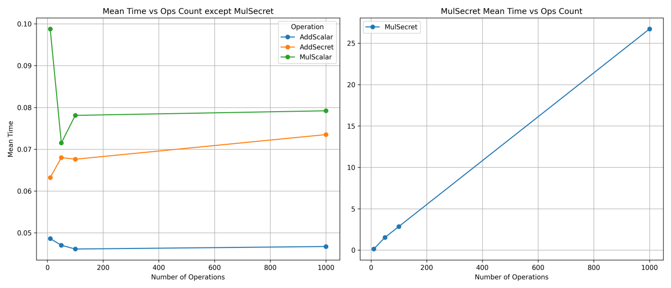

Fig. 1: Time mean vs number of operations

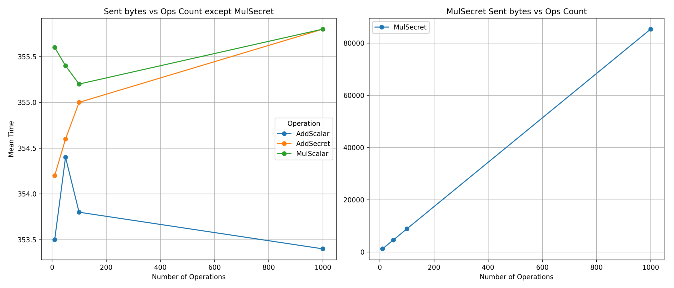

Fig. 2: Sent bytes vs number of operations

## V. Performance evaluation

For performance evaluation, we developed a program that automatically generates circuits based on the specified size. It allows us to vary the number of parties, multiplications, additions, and repetitions to ensure reliable results.

This is the structure of the results obtained (I and II):

| Op | Parties | Ops | Client | Time | Mean ± Std |
|---|---|---|---|---|---|
| AddScalar | 2 | 10 | Party1 | 0.0590 | 0.0965 ± 0.0934 |
| AddScalar | 2 | 10 | Party0 | 0.2574 | 0.0965 ± 0.0934 |
| AddScalar | 2 | 10 | Party0 | 0.0353 | 0.0965 ± 0.0934 |
| AddScalar | 2 | 10 | Party1 | 0.0343 | 0.0965 ± 0.0934 |

TABLE I: Execution time for various operations and parties.

| Op | Parties | Ops | Client | Sent | Sent Mean | Recv | Recv Mean |
|---|---|---|---|---|---|---|---|
| AddScalar | 2 | 10 | Party1 | 141.0 | 142.2 | 35.0 | 34.5 |
| AddScalar | 2 | 10 | Party0 | 143.0 | 142.2 | 34.0 | 34.5 |
| AddScalar | 2 | 10 | Party0 | 143.0 | 142.2 | 34.0 | 34.5 |
| AddScalar | 2 | 10 | Party1 | 142.0 | 142.2 | 35.0 | 34.5 |

TABLE II: Data transmission per operations and parties.

We can observe from the mean time graph (see Figure 1) that when Beaver triples are not used for secret multiplication, operations are executed very quickly. As the number of operations increases, the mean execution time grows only slightly and eventually plateaus.

In contrast, when using Beaver triples for multiplication, the performance differs significantly due to the additional communication overhead. The graph exhibits a linear growth pattern, indicating that these operations are just repeated.

Regarding the bytes sent and received (see Figure 3), we observe that communication overhead impacts only Beaver multiplication operations, where growth is also linear.

Finally, when varying the number of parties (Figure 4), the effect on time becomes significant mainly for the multiplication of secrets that use a beaver, as it needs more time to share data.

## VI. Application

This application uses Secure Multiparty Computation (SMC) to evaluate friendship compatibility based on private traits such as love for pizza, political views, sleep habits, and emotional tolerance. Each participant inputs personal values, and the protocol computes a shared compatibility score without revealing individual inputs.

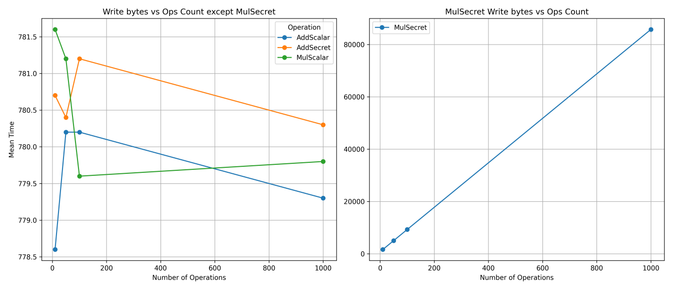

Fig. 3: Received bytes vs number of operations

The goal is to allow honest, sensitive input while protecting privacy. This could be useful in social apps, team-building tools, classrooms, or anonymous surveys, where participants may hesitate to share preferences openly.

We assume an *honest-but-curious* adversarial model: all parties follow the protocol but may try to infer others' inputs from the output. While SMC hides inputs during computation, observing the final score can still leak information—especially when inputs come from small domains. For example, in `test_friendship3`, knowing one's own values and the output could reveal the other party's political view.

To reduce such leakage, exact outputs can be replaced with coarse labels (e.g., "Compatible"), or randomized with small noise. Limiting repeated runs and using wider input ranges also help mitigate inference risks. These techniques preserve privacy while still enabling useful, socially aware computations.

## References

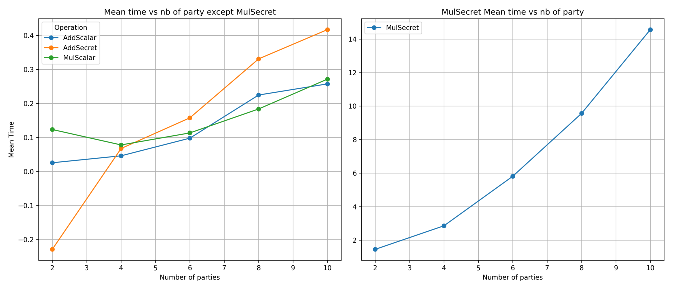

Fig. 4: Mean Time in function of nb of parties