# CS-523 SMCompiler Report

Florian Delavy , Endrit Vorfaj

***Abstract*—Secure Multiparty Computation (SMC) enables multiple parties, each having access to fragments of the data we want to keep private, to jointly compute a specific outcome. This fragment is referred to as a "Share" in our work and a "Share" can either be a "Secret" or it can be a "Scalar", which needs not be kept private. By combining their shares, the final result is computed without revealing the nature or content of their inputs or any other confidential information involved in the process. We build our SMC framework under an Honest-but-Curious threat model, which assumes that parties do not deviate from their intended behaviour in the protocol. Additionally, we provide performance metrics of the circuit in terms of the number of parties and operations, along with a real-world use case of the implemented SMC framework.**

## I. Introduction

In this project, we both collaborated closely on all aspects. However, each individual focused on specific areas while getting constant support from the other. F.D primarily contributed to the `expression` class, addition and multiplication of scalars (along with interactions involving their sub-expressions) in the `smc_party` class, and performance measurement tests in `metrics`. On the other hand, E.V concentrated on beaver multiplication within `smc_party`, `ttp` and `secret_sharing`, as well as code for measuring time and bytes sent and received. Both team members jointly conducted tests, developed the application, and authored the report.

## II. Threat model

The threat model of our SMC framework considers all parties to be honest-but-curious (HBC), meaning that they will follow the protocol's instructions but are interested in obtaining information from the messages exchanged during the computation. The threat model assumes that an adversary may attempt to learn information about the inputs and intermediate computations of other parties, however, the HBC model has limitations as it is not capable of sending falsified messages. Even when compared to a passive adversary, the HBC adversary is still more restricted as it cannot intercept messages from arbitrary communication channels, and can only receive messages that are intended for it. To guarantee that the computation is secure and that no party can obtain information about the other parties' inputs or intermediate computations, the SMC framework uses cryptographic techniques. Secret sharing is one such technique, in which each party shares their input with the other parties in such a way that no one party has the entire secret. This ensures that no party has full knowledge of another party's input. It also assumes that the communication channels between parties are secure, and messages cannot be intercepted or altered by an external attacker. This assumption in our case is limited to our SMC model as all communication between parties happens through a module that was provided and we have no proof of its security.

## III. Implementation details

In our implementation of the SMC we start by first implementing the `Expression` module where we define the abstract syntax tree for arithmetic expressions used in the SMC protocol. It has sub-classes for "Secret" and "Scalar" variables and it defines the representation of the operations we need in our SMC such as addition, subtraction, and multiplication. The `secret_sharing` module offers a secret sharing scheme that splits a secret value into shares and distributes them among multiple parties. This module provides a `Share` class that represents a single share of the secret value or a scalar. Besides defining the protocols for our three operations, this module also provides functions for splitting a "Share" and reconstructing the original secret value from the shares. The `SMCParty` is the main class that implements an SMC client. It takes as input a client ID, the host-name and port of the server, a protocol specification, and a dictionary of secret values to be used in the protocol. The class makes use of the `secret_sharing` class for splitting secrets into shares and then it sends them to other participants. Depending on whether the `Share` is a scalar or a secret, the reconstruction is done accordingly by the use of the dictionary and the `Communication` class. It also implements methods for processing expressions using the visitor pattern, and for performing multiplication using the Beaver triplet protocol generated by a trusted third party. The class communicates with the server and other clients using the Communication class which was provided for this project.

## IV. Performance evaluation

To evaluate the performance of our SMC implementation, we collected data on both communication (bytes sent and received) and computation time.

| Nr. | Runtime | Byte Sent | Byte Received |
|---|---|---|---|
| 1 | 15.26 (2.01)<br>10246 (17.89) | 15.68 (7.51)<br>15.72 (7.59) | 62.72 (7.51)<br>62.88 (7.60) |
| 10 | 52,68 (78.9)<br>10245 (19.95) | 15.6 (7.34)<br>15.72 (7.59) | 62.4 (7.34)<br>62.88 (7.60) |
| 100 | 16.68 (4.35)<br>10230 (28.10) | 15.72 (7.59)<br>15.60 (7.35) | 62.88 (7.60)<br>62.40 (7.37) |
| 500 | 17.49 (3.04)<br>10252 (19.47) | 15.64 (7.43)<br>15.64 (7.43) | 62.56 (7.43)<br>62.56 (7.46) |

TABLE I: Addition of Scalars.

| Nr. | Runtime | Byte Sent | Byte Received |
|---|---|---|---|
| 1 | 241 (40.33)<br>17570 (994.92) | 54.8 (49.7)<br>54.96 (49.83) | 56.12 (12.46)<br>146.52 (12.44) |
| 10 | 89.49 (83.38)<br>42960 (187.51) | 273 (2.56)<br>273.92 (1.8) | 364.56 (2.00)<br>365.12 (1.92) |
| 100 | 526.6 (94.37)<br>338167 (912.68) | 2461.6 (6.02)<br>2462 (4.75) | 2553.28 (6.43)<br>2553 (4.70) |
| 500 | 1783.85 (112.86)<br>1649843 (2676.92) | 12186 (13.46)<br>12192 (12.33) | 12277.08 (12.19)<br>12283 (11.66) |

TABLE II: Addition of Secrets.

| Number | Runtime | Byte Sent | Byte Received |
|---|---|---|---|
| 1 | 18.54 (3.56)<br>10950 (17.03) | 15.68 (7.51)<br>15.68 (7.51) | 62.72 (7.52)<br>62.72 (7.52) |
| 10 | 32.07 (28.08)<br>10246 (18.44) | 15.72 (7.60)<br>15.64 (7.43) | 62.88 (7.62)<br>62.56 (7.43) |
| 100 | 40.60 (28.15)<br>10254 (16.88) | 12 (0.0)<br>12 (0.0) | 48 (0.0)<br>48 (0.0) |
| 500 | 194.17 (94.75)<br>10272 (18.33) | 12 (0.0)<br>12 (0.0) | 48 (0.0)<br>48 (0.0) |

TABLE III: Multiplication of Scalars.

| Number | Runtime | Byte Sent | Byte Received |
|---|---|---|---|
| 1 | 233.02 (2.53)<br>17589 (993.27) | 54.76 (49.73)<br>54.4 (49.4) | 146.08 (12.69)<br>145.12 (12.46) |
| 2 | 491.95 (172.66)<br>42488 (954.45) | 139.60 (60.75)<br>139.88 (60.53) | 413.08<br>413.84 (15.44) |
| 5 | 510 (311.67)<br>116532 (207.26) | 395.40 (2.7)<br>395.08 (1.49) | 1217.04 (2.68)<br>1216.48 (3.26) |
| 10 | 1445.44 (414.19)<br>245669 (325.26) | 819.80 (2.88)<br>820 (2.66) | 2551.16 (5.52)<br>2552 (4.39) |
| 20 | 2669.29 (978.9)<br>503353 (500.43) | 1172.36 (4.97)<br>1672 (4.51) | 5229.64 (8.15)<br>5229 (7.61) |

TABLE IV: Multiplication of Secrets.

| Number | Runtime | Byte Sent | Byte Received |
|---|---|---|---|
| 2 | 50.15 (64.76)<br>19737.52 (1089.87) | 121.8 (1.13)<br>120.9 (1.79) | 137.00 (16.69)<br>136.10 (16.22) |
| 5 | 554.68 (146.45)<br>44189.81 (1079.37) | 139.76 (64.05)<br>139.88 (60.78) | 437 (25.05)<br>438.08 (24.82) |
| 10 | 493.26 (97.21)<br>85657.25 (510.73) | 146.12 (110.50)<br>145.94 (110.69) | 904.84 (19.83)<br>902.28 (19.48) |
| 20 | 553.76 (43.80)<br>125843.22 (739.72) | 147.66 (145.68)<br>152.82 (150.77) | 1358.81 (17.13)<br>1361.89 (16.78) |
| 25 | 1042.76 (47.35)<br>207182.45 (1555.63) | 149.6 (198.28)<br>150.12 (199.38) | 2280.68 (14.20)<br>2278.27 (13.73) |

TABLE V: Varying number of parties.

We conducted five experiments to evaluate the performance of the SMC under different scenarios involving varying numbers of additions, multiplications, and parties. The experiments were run on two different machines to observe the effect of hardware on the results, and were limited to five iterations due to time constraints (approximately 7 hours on the slower machine). The results are presented in five tables.

Table I shows the runtime of the SMC with different numbers of additions of scalars using five clients. The runtime (in milliseconds) remains relatively constant as each party adds the scalars locally. However, the hardware has a significant impact on performance, with the faster (blue) machine running approximately 190 times faster than the slower machine (red).

Table II presents the results of testing for addition of secrets using the same set of numbers (1, 10, 100, 500). This test takes longer as secrets need to be sent to other participants. The faster machine performs approximately 60 times better than the slower machine. Notably, the amount of data transferred between the machines is almost the same, indicating that the hardware is slow but not faulty.

Table III and Table IV evaluate the runtime for scalar and secret multiplication, respectively. Scalar multiplication has a similar runtime to addition, but secret multiplication takes significantly longer due to the use of Beaver triplets. The number of bytes exchanged increases linearly with the number of multiplications.

Finally, Table V varies the number of parties involved in the computation. The same expression, including all available SMC operations, is used. As expected, adding more parties increases the runtime, but it remains cheaper than secret multiplication.

## V. Application

Secure Multiparty Computation (SMC) can be used to securely share a symmetric session key among multiple parties. The parties use an expression and random secrets to compute the result, which is used as the session key to encrypt messages that can only be decrypted by the intended recipients. Since the secrets are random and the goal is to send private messages among the clients, passive adversaries can only learn the result of the expression and maybe the random secrets, which are not useful to them. In the case of aggressive adversaries, they can only give incorrect results of the expression, which is easily noticeable when attempting to encrypt and decrypt values with an incorrect key.

Regarding adversaries outside the clients, there could be potential privacy leaks that are not covered by SMC. To mitigate this, additional security measures such as encryption and authentication could be employed. With a simple implementation, any circuit would allow the generation of a shared key as long as each participant knows it.