CS523 Project 1 Report

Knuchel Benoit (270320), Mathez Bradley (260413)

***Abstract*—We implemented a SMC protocol that supports expressions made of additions, subtractions and multiplications of integers. We then made a performance evaluation and proposed an application.**

I. Introduction

Secure multi-party computation (SMC) is a strong protocol which allows a group of people to jointly compute a result without publicly releasing their secret inputs. The main difference with traditional encryption methods is that it does not only protect parties' secrets against external attackers but also against other participants.

This project aims to implement an SMC engine that resists against passive attacks using Python3. We will consider additions, subtractions and multiplications of integers. We will use the *Beaver triplet protocol* to implement the multiplications of secrets. We were also asked to find an application for which our SMC solution could be useful and to code a proof of concept.

II. Threat model

We want to implement a solution that resists against passive attacks (i.e. semi-honest adversary). Within this assumption, we assume that the attacker follows the protocol and only tries to infer information. As we split a secret into shares, then an attacker has to corrupt every participants (each participant contributes to the computation with some secrets/scalars) in order to learn the secrets, to be able to evaluate the circuit and retrieve the final result.

The types we may encounter for an attacker are eavesdroppers, curious participants or attackers that corrupts $p-1$ parties where p is the total number of participants. The main goal is to achieve the same security level as an ideal setting (i.e. using a trusted third party). We want the parties to not learn more than what they might have learned in the ideal setting.

III. Implementation details

First of all, we decided to set q (i.e. the modulo used to compute shares) to $q = 2^{20} = 1'048'576$ because it is the first power of 2 that is bigger than 1 million and we thought that it was big enough to deal with the problems proposed in tests and in our application.

In order for a participants to use the SMC system, the party has to generate the shares for his secrets, process the expression recursively (i.e. one operation by one), broadcast the result (which is one of the result's shares), retrieve the others' shares and combine them to obtain the final result.

The main part is the expression processing. The *additions* and *subtractions* are handled the same way. We process the expression of each components and return the addition (resp. subtraction) of the two results.

If we are dealing with *scalars*, we have to add it to a single share or to multiply every shares with it depending on the operation. For this and to deal correctly with multiplication of secrets, we have to check recursively if the operands contain a secret or not (we called the function `has_secret`). Then we can call `process_expression` with a flag that represents if the operand contains a secret or not. As we only update this flag when we encounter a multiplication, we can also use it to check if we are in a multiplication. This allows to asses the problem to deal with scalars and to decide to return `Share(expr.value)` or `Share(0)`.

Finally the big part is to deal with *multiplications* because we need to know if both operands contain at least one secret in order to decide if we must use the Beaver triplet protocol. We firstly call `has_secret` it on each operand. We will discuss the two cases (i.e. both operands contain at least one secret, or not).

Let's start if they both contain at least one secret. Then we process the operands' expression with the flag set to *True*. Next, we generate Beaver shares (i.e. $[x - a]$, $[y - b]$ and $[c]$) using the Beaver triplet protocol described in class. To conclude, we combine the shares to compute the final result (while being careful that only one participant subtracts $(x - a)(y - b)$).

For the other case, we simply return the multiplication of the processed operands' expression with the flag set to true if one of the two contains at least one secret or if the current call to `process_expression` was performed with the flag set to true.

In the case where we need to perform an operation on a single party (e.g. add a scalar), then we execute the computation on the first participant only.

IV. Performance evaluation

In order to evaluate the performance of our application we measured the effect of the number of parties on the cost, the effect of the number of additions on the cost and the effect of the number of multiplications on the cost. Here the cost is defined as either the computation time or the number of bytes sent and received by a party. Note that we split the addition (respectively multiplication) in two tests, one where we add (respectively multiply) only scalars and one with only secrets. As the subtractions are handled in the exact same way as additions (i.e. only the operation changes), we assumed that the metrics are similar. To obtain Tables I and II, we fixed the number of participants at 5. For table III, we defined a simple circuit that grows with the number of participants : we used

$f(a, b, c, ...) = K + a + b + c + ...$ (i.e. additions of secrets with a scalar as the first component.) For every case we repeated the experience 15 times and then averaged the results.

Time[s]	n=10	n=100	n=500	n=1000
Add. of secrets	5.2	5.3	7.4	9.6
Add. of scalars	5.1	5.1	5.2	5.2
Mult. of secrets	6.6	23.7	101	193
Mult. of scalars	5.1	5.1	5.4	5.9

TABLE I: Average computation time[s] with a growing number of operations[n]

Bytes	n=10	n=100	n=500	n=1000
Add. of secrets	113	549	2433	4781
Add. of scalars	32	33	33	34
Mult. of secrets	1009	10450	52390	104830
Mult. of scalars	34	35	35	35

TABLE II: Average number of bytes sent/received per client with a growing number of operations[n]

	p=3	p=10	p=50	p=100
Time[s]	5	5.6	14.1	48
Bytes	20	172	885	1776

TABLE III: Average time[s] and number of bytes sent/received per party with a growing number of participants[p]

Time[s]	n=10	n=100	n=500	n=1000
Add. of secrets	0.13	0.07	0.25	0.73
Add. of scalars	0.01	0.05	0.01	0.06
Mult. of secrets	0.19	3.48	3.54	9.03
Mult. of scalars	0.07	0.09	0.03	0.28

TABLE IV: Standard deviation of computation time[s] with a growing number of operations[n]

Bytes	n=10	n=100	n=500	n=1000
Add. of secrets	2.38	15.09	77.72	177
Add. of scalars	1.11	1.29	1.58	1.95
Mult. of secrets	6.59	24.2	102.1	176.3
Mult. of scalars	1.78	2.04	2.07	2.28

TABLE V: Standard deviation of number of bytes sent/received per client with a growing number of operations[n]

	p=3	p=10	p=50	p=100
Time[s]	0.09	0.22	0.56	2.58
Bytes	1.29	2.02	4.39	6.87

TABLE VI: Standard deviation of time[s] and number of bytes sent/received per party with a growing number of participants[p]

In the case of addition of scalars, the communication cost is constant but the computation cost is growing linearly but very slowly. The constant number of bytes sent and received is because one party computes the addition and then sends the result to the other parties. With the secrets, the results are a bit different because each party now has to hide their secrets and send parts of it to other parties. This explains the linear growth of communication cost. Since the computation cost highly depends on communication, it also increases in contrary to the scalar case.

The case of multiplication with scalars is similar to the one of addition, the communication cost is the same but we can see that the computation cost grows a bit faster with the number of operations. This is explained by the fact that multiplying two numbers takes more time than adding them. The computation time of multiplications of secrets takes much more time but is also linear. As for the addition of secrets, the number of bytes sent greatly impacts the time and in this case the parties need to communicate a lot. This is mainly due to the Beaver triplets which needs a lot of exchange between parties and server. From Table I and IV, the time needed to multiply 1000 secrets between 5 parties is in average 193 seconds with a standard deviation of 9.03 seconds.

From Table III, we can see that the number of bytes communicated grows linearly (hence also the computation time) with the number of participants.

Also Tables IV, V and VI represent the standard deviations of our tests. We can see that they are relatively small compared to the averaged results.

V. Application

A. Application description

Our application is located at the end of "test_integration.py" and is named "test_application".

It represents three families of friends that want to go together for three different trips during the vacations. Each has to put a certain (secret) amount of money on the shared electronic wallet. Also each family has to vote (secretly) whether or not they agree to take the plane to destinations.

They want to compute the total amount of money they dispose of for the vacations and they will only fly to destinations if everyone agrees. For that latter part, we used a multiplication of secrets which have been set to 1 (agree) or 0 (discard).

The families indicate their budget for a single week of each trip. Then the program checks their reservation and extracts the duration (in weeks) of their trip and multiplies their budget by this amount.

The group of three families take out an insurance for their trip which costs 50CHF per trip. However, since they are a group of people, they are entitled to a 200CHF discount on their first trip.

B. Possible privacy leakages

A possible leakage in our application may be that two families are really close to each other and they share their respective inputs for the program. Then, they are able to compute the total amount of money or acquire information about the vote of the last family. This easy leak is due to the fact that there are only three families in the example. However it is a real problem if $p - 1$ families share their inputs.