CS-523: SMCompiler Project

The goal of this project is to implement and apply a secure multi-party computation engine (secure MPC, or SMC) in a semi-honest (passive) adversarial setting using the Python 3 programming language. You will implement an SMC framework that works for generic arithmetic circuits assuming the existence of a trusted third party.

We provide you with a project skeleton and tests:

https://github.com/spring-epfl/CS-523-public

Deliverables

By the due date specified on Moodle you must submit:

1. A 2-page report in IEEE 2-column format based on the provided report skeleton. The LaTeX template for your report can be found in https://github.com/spring-epfl/CS-523-public.

2. A `zip` or `tar.gz` code archive that contains a working implementation of the different parts of the project in Python and a README file.

Suggested Timeline

You have five weeks in total for this project. We suggest the following timeline:

- Week 1: Understand the project and the skeleton code (see Part 1.1).

- Week 2-3: Implement the first part of the SMC framework (see Parts 1.3, 1.4, and 1.5).

- Week 3-4: Implement the multiplication in the trusted third-party setting (see Parts 1.6 and 2).

- Week 5: Describe an application, write the report, and prepare the submission.

1 SMC Framework

In this project, you will implement the necessary components for secure multi-party computation on arithmetic circuits assuming the existence of a trusted third-party and additive secret sharing. Recall that additive secret sharing allows one to secretly share a secret into multiple sub-secrets, one for each entity, such that only the knowledge of the sum of all sub-secrets yields the original secret. We denote by $[x]$ the additive secret sharing of a value x among N parties. More precisely, $[x] = x_0, ..., x_{N-1}$ such that $x = \sum_{k=0}^{N-1} x_k$.

We consider arithmetic circuits made of additions, addition of a constant, subtractions, multiplication by constants, and multiplications.

1.1 Getting started

Retrieve the code from https://github.com/spring-epfl/CS-523-public. It contains a skeleton for your implementation of the main functions and for a set of unit tests. You need to implement a SMC client, a trusted parameter generator (for the Beaver triplet scheme), the secret-sharing scheme, and the engine for creating arithmetic circuits (we also call them arithmetic *expressions* in the skeleton). You are free to change the skeleton and design the system in your own way as long as your code passes the test suite in `test_integration.py` *without any changes to the tests*.

Additionally, for your convenience, the code contains utilities that enable communication between SMC parties. Please get familiar with the code and how it is structured, and check the README and comments in each of the files for additional instructions and suggestions for your implementation.

Name	Parties	Function
test_suite1	3	$f(a, b, c) = a + b + c$
test_suite2	2	$f(a, b) = a - b$
test_suite3	3	$f(a, b, c) = (a + b + c) * K$
test_suite4	3	$f(a, b, c) = (a + b + c) + K$
test_suite5	3	$f(a, b, c) = (a * K0 + b - c) + K1$
test_suite6	4	$f(a, b, c, d) = a + b + c + d$
test_suite7	3	$f(a, b, c) = (a * b) + (b * c) + (c * a)$
test_suite8	5	$f(a, b, c, d, e) = ((a + K0) + b * K1 - c) * (d + e)$

Table 1: Description of the test circuits.

1.2 Testing the code

We provide you with high-level integration tests for all types of the circuits that your implementation has to pass (See Table 1). All the existing tests in `test_integration.py` *must* pass without any modifications to the `test_integration.py` file.

In addition to the main test suite, you can optionally implement additional tests for each of the sub-components of the protocol (the SMC client, the trusted parameter generator, the secret-sharing scheme, and the engine for creating arithmetic circuits). We provide you with skeletons of such unit tests that you can adapt to your own class structure and interfaces. See the README of the skeleton repo for the details. In this project, implementing the unit tests is optional and does *not* affect your grade on its own. It will, however, make your life easier, and enable to prepare for the next project in which the tests will be graded.

To create a new test suite, create a file with a name beginning with `test_` with an extension `.py`. You can name the rest of the file as you prefer, in this example we went with `test_operations.py`. Write your tests in the file as you see fit (you can take inspiration from the test suite we provided if needed). Then, you can run the tests you wrote with the command

```
python3 -m pytest test_operations.py
```

1.3 Additive SMC protocol

Starting from the skeleton code, we suggest you first implement an SMC protocol for executing addition operations.

An addition operation takes two inputs and returns its sum. Recall that for a secret sharing of $[x]$ and $[y]$, a party can just locally compute:

$$[x + y] = [x] + [y]$$

For this, you have to implement the secret sharing scheme, the creation of additive arithmetic circuits, and the SMC client code that runs the appropriate protocol.

We suggest that you also write unit tests for the new operation along with its implementation, or even before you implement it.

1.4 More operations based on addition

Now that you have a working code running SMC for addition operations, you can add more operations:

- Subtractions: $[x - y] = [x] - [y]$

- Scalar multiplications: $[k \cdot x] = k \cdot [x]$

As before, we suggest that you also write unit tests for the new operation along with its implementation, or even before you implement it.

1.5 Addition of a constant

Afterwards, you can implement another operation that can be computed locally by each participant: the addition of a constant.

The SMC procotol for computing this operation is more involved than the standard addition operation as not all parties have the same role. For a constant k, the secret-sharing of the constant addition is:

$$[x + k] = \{x_0 + k, x_1, ..., x_{N-1}\}$$

Thus, only one of the participants is adding the constant.

1.6 Multiplication using the Beaver triplet protocol

Finally, you need to implement the multiplication protocol using the Beaver triplet scheme. This scheme makes use of "blinding" values a, b, and c sampled randomly such that $ab = c$. Those values are also additively shared and called Beaver triplets $[a]$, $[b]$, and $[c]$.

Given inputs $[x]$ and $[y]$ that are being multiplied, each party performs the following operations:

1. Locally generates the share $[x - a]$ and broadcasts (publishes) it.

2. Locally generates the share $[y - b]$ and broadcasts it.

3. Reconstructs $(x - a)$ and $(y - b)$ from the published shares of these values.

4. Locally computes:

$$[z] = [c] + [x] \times (y - b) + [y] \times (x - a) - (x - a)(y - b)$$

NOTE: Whereas the term $-(x - a)(y - b)$ is known by all the parties, it must be added locally by only one party, otherwise the result will be off by $(n - 1) \cdot (-(x - a)(y - b))$. Therefore, a multiplicative operation also contains an addition-of-a-constant operation.

One can verify that $[z]$ is indeed the additive secret sharing of $[xy]$. Since the values a, b, and c are sampled uniformly at random, $(x - a)$ and $(y - b)$ are protecting the values of x and y providing information-theoretic privacy for the inputs.

To implement this protocol, as before, you need to add the ability to specify multiplication operations within circuits, and the appropriate SMC-client functionality that executes the protocol above. Additionally, you also need to implement the trusted third party that generates the Beaver triplets (a, b, c) and provides each SMC protocol party with their respective share of $[a]$, $[b]$, and $[c]$ for all the multiplications in the circuit. Note that each multiplication operation requires a new triplet.

As previously, we suggest that you also write unit tests for the new operation along with its implementation, or even before you implement it.

1.7 Performance Evaluation

After ensuring that all the above protocol components work correctly, you have to evaluate the performance of your implementation in terms of both communication (bytes sent and received) and computation costs (computation time).

Your evaluation has to show:

1. The effect of the number of parties on the costs

2. The effect of the number of addition operations, and, separately, additions of scalars

3. The effect of the number of multiplication operations, and, separately, scalar multiplicatons

For this, you need to design experiments in which you vary the corresponding parameters of the SMC protocol (e.g., to check the effect of the number of additions you could evaluate circuits with 10, 100, 500, 1000 addition operations) and report the measurements of the costs for each value of the parameter. The costs should be measured for each participating party or a third party. The measurements have to be reported along with the statistical information (e.g., mean and standard deviation in a table, or the mean value and errorbars in a plot).

You should think about the following questions before reporting your performance evaluation. Does your performance evaluation depend on the hardware that you are running? What about the way you set up your testing environment? How many repeat measurements do you need to compute the standard error?

2 Custom Application

Having implemented all the protocols in the previous section, you should obtain a versatile implementation of an SMC engine that is capable of handling a wide range of expressions.

For the final part of this project, you should come up with a potential use case of SMC and a circuit that implements the desired computation. Your circuit should contain at least one kind of each of the operations you implemented and require multiple parties to participate in the computation. In addition to the implementation of the actual circuit you should add tests that validate the correct implementation of the circuit. You can either add these tests in the test_integration.py file or in a new file using utilities from test_integration.py.

Project Requirements

Your report should answer the following questions:

- What is the aim of the framework you have implemented?

- What is its adversarial model?

- How did you evaluate the performance of the framework? Detail your experimental design. You need to report a fine-grained evaluation of communication and computation costs. Each reported measurement must include both the mean and standard error of the mean.

- What is the use case for your custom application? What is the adversarial model in this scenario? What could this computation be useful for?

- Is there any privacy leakage if one observes the output of the SMC protocol in your custom application? If yes, discuss possible solutions to mitigate these risks.

The code that you deliver must:

- Pass all tests in the test_integration.py suite without any modifications to these tests.

- Follow common coding practice like providing good documentation, having comments, having understandable variable names, etc.