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`

Note: Forks of the project code repository must remain private.

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 Section 1.1).
- Week 1–2: Implement the first part of the SMC framework (see Section 1.3, Section 1.4, and Section 1.5).
- Week 3–4: Implement the multiplication in the trusted third-party setting (see Section 1.6, Section 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, see Appendix A for more details). You are free to 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 file.*

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.

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

A Demystifying the Expression Module in the Template

Once you have familiarized yourself with the code template, you might have some questions about the role of the "expression" module. This section aims to clarify these questions and describe expressions in more technical detail.

In order to execute an SMC protocol, all parties need to agree on the arithmetic circuit they are computing. Thus, in your implementation you need to be able to represent arithmetic circuits in a convenient way. This is what the expression module template is for. An expression is a representation of an arithmetic circuit that can be passed around and is easy to work with.

A.1 Expressions cannot be immediately interpreted

An arithmetic circuit is a function, not a value of a function. For example, $f(a, b) = a+b$. When an expression representing $f(a, b)$ is constructed, you cannot compute its concrete output: the values of a and b are simply not known at construction time. Thus, in an implementation of expressions, you have to represent the structure of $f(a, b)$ in an abstract way, not its concrete output value.

A.2 Expressions are abstract syntax trees

We now know that an expression should somehow encode the structure of an arithmetic circuit. How can this be done? In fact, the only way to do this in a way that is compatible with our integration tests is to build an abstract syntax tree. An abstract syntax tree is a tree-based data structure that is used internally by compilers and interpreters to represent a structure of an expression in a programming language. This is exactly our case: We have a special language of arithmetic circuits, and we want to be able to represent them in a convenient way. Consider a circuit $f(a, b, c) = a \cdot b + c$. It can be represented using the following abstract syntax tree in Fig. 1.

The intermediate nodes of the tree (circles) represent operations, and the leaf nodes (squares) contain the terms—the most basic atoms—of our language. In this example, the terms are variables such as a, b, c.

How can this be implemented in Python? The simplest way to do so is the following:

```python
# Intermediate tree node representing addition operation
class AddOp:
  def __init__(self, a, b):
    self.a = a
    self.b = b

# Intermediate tree node representing multiplication operation
class MultOp:
  def __init__(self, a, b):
    self.a = a
```

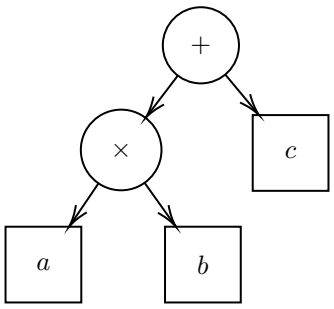

Figure 1: Abstract syntax tree for $f(a, b, c) = a \cdot b + c$.

```
    self.b = b

# Leaf node representing a variable
class Variable:
  def __init__(self, name=None):
    self.name = name
```

Using this implementation we can represent $f(a, b, c) = a \cdot b + c$ as:

```
AddOp(MultOp(Variable("a"), Variable("b")), Variable("c"))
```

This basic implementation is already sufficient to represent any arithmetic circuit in Python. However, an implementation like this would not pass our integration tests as we expect a slightly different interface for the creation of circuits. In fact, we expect a more intuitive way to create circuits that the Python language supports: we want to be able to write an expression in natural notation. For example, we want to simply write `a * b + c` to represent $f(a, b, c) = a \cdot b + c$. How to enable this way of constructing expressions? First, we have to leverage Python's operator overriding using `__add__` and `__mul__` methods. Second, we have to use an abstract base class for all expressions:

```
# This is the base class for all expressions.
# Similar to an abstract class in Java or C++.
class Expression:
  def __add__(self, other):
    return AddOp(self, other)

  def __mul__(self, other):
    return MultOp(self, other)

# Intermediate tree node representing addition operation
# (Note it is now an instance of Expression)
class AddOp(Expression):
  def __init__(self, a, b):
    self.a = a
    self.b = b

# Intermediate tree node representing multiplication operation.
# (Note it is now an instance of Expression)
class MultOp(Expression):
  def __init__(self, a, b):
    self.a = a
    self.b = b

# Leaf node representing a variable.
# (Note it is now an instance of Expression)
class Variable(Expression):
  def __init__(self, name=None):
    self.name = name
```

Now, we are able to define $f(a, b, c) = a \cdot b + c$ using the natural notation:

```
Variable("a") * Variable("b") + Variable("c")
```

This is not the only way to implement the functionality, but rather the most idiomatic (standard) for Python. This example shows a construction of an abstract syntax tree that represents arithmetic operations on some "variables." In this example, Variable is the most basic term of a language. In the case of our SMC protocol, the terms are a Scalar and a Secret, thus the tree implementation has to be slightly adapted.

A.3 How to use expressions in a run of the protocol

To run the protocol, each party has to execute an appropriate sub-protocol (addition, scalar addition, scalar multiplication, multiplication using the Beaver scheme) for each operation in the expression. How do we traverse expressions to run a sub-protocol for every operation?

To traverse an expression, we run a tree traversal algorithm: starting from the root node, we recursively visit each sub-expression until we reach the leaves of the tree. An idiomatic way to do so in Python uses the so-called "visitor" pattern:

```
def traverse(expr):
  if isinstance(expr, AddOp):
    # Evaluate the addition, and
    # traverse further by calling traverse(expr.a) and traverse(expr.b)
    return value

  if isinstance(expr, MultOp):
    # Evaluate the multiplication,
    # traverse further by calling traverse(expr.a) and traverse(expr.b)
    return value

  if isinstance(expr, Variable):
    # Handle the base of recursion.
    # For example, retrieve the value assigned to the variable from somewhere.
    return value
```

In our case, however, we do not have Variables as leaves — we have Secrets and Scalars that are treated differently, and we do not simply "evaluate" additions or multiplications — rather we run SMC sub-protocols.