Problem 4 Secure Multiparty Computation Protocol Design

Four different parties (Alice, Bob, Chris, and David) locally hold four different vectors, respectively (10 integers in each vector).
• Alice holds: $V_a = [a_1, a_2, \cdots, a_{10}]$.
• Bod holds: $V_b = [b_1, b_2, \cdots, b_{10}]$.
• Chris holds: $V_c = [c_1, c_2, \cdots, c_{10}]$.
• David holds: $V_d = [d_1, d_2, \cdots, d_{10}]$.
Design a cryptographic protocol to securely sum all four vectors: $V = V_a + V_b + V_c + V_d$, and find the maximum value (out of the 10 entries) in V.

Hints:
• You can use the combination of Homomorphic Encryption (e.g., Paillier's Cryptosys- tem), Garbled Circuit (e.g., Fairplay) and Permutation to solve this problem.
• Sum V should not be disclosed to any party. The maximum value in V will be the only output.
• If necessary, two-party Fairplay can also be used to securely compare multiple values by executing multiple comparisons.
• Only using Garbled Circuit (e.g., Fairplay) may not be computationally practical.
• Try to reduce the information leakage in the protocol as much as possible.

Tasks:
1. Protocol Design (write the pseudocode in the report).
2. Implementation of the Protocol.

Submission requirement:
(1) a report including the protocol design,
(2) implementation and testing details in the report (bonus), and
(3) source code files (bonus) – all named with the prefix "hw3-4-" (e.g., hw3-4-report.pdf).

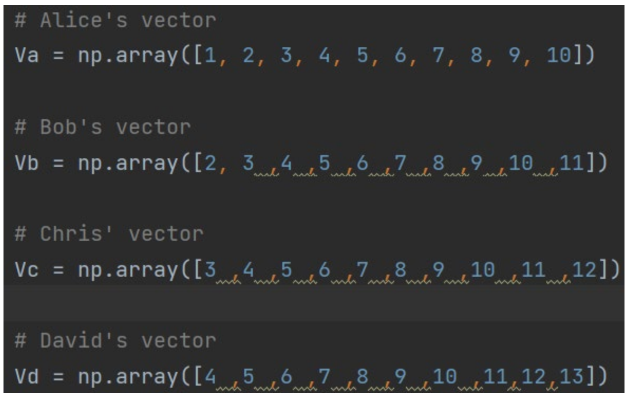
Task 1 Protocol Design

There are at least two ways to securely sum all four vectors.

Way 1 Use Homomorphic Encryption (Paillier's Cryptosys- tem) + Secure Multiparty Computation
We can use a combination of homomorphic encryption and secure multiparty computation to securely sum the four vectors and find the maximum value. Here is the protocol I designed (The code implementation script is hw3-4-code-HE.py. When you run the script which is python code, make sure your system has installed phe, numpy libriries):

Step 1. Define 4 vectors as four parties. Each party generates a random vector of 10 integers, as follows:
Alice generates Va = [a1, a2, ..., a10].
Bob generates Vb = [b1, b2, ..., b10].
Chris generates Vc = [c1, c2, ..., c10].
David generates Vd = [d1, d2, ..., d10].

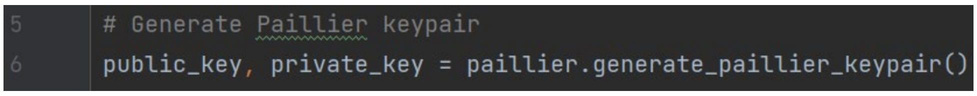
Key code:



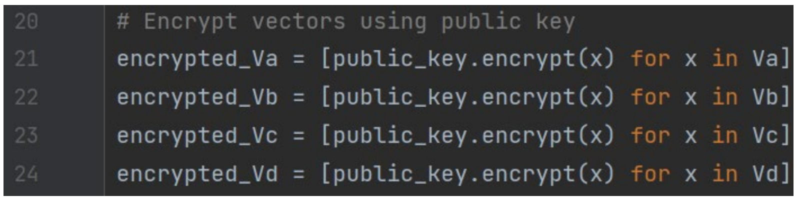
```
# Alice's vector
Va = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Bob's vector
Vb = np.array([2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

# Chris' vector
Vc = np.array([3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

# David's vector
Vd = np.array([4, 5, 6, 7, 8, 9, 10, 11, 12, 13])
```

Step2. Each party encrypts its vector using a homomorphic encryption scheme that supports the addition of ciphertexts with the Paillier cryptosystem. Each party encrypts its vector as follows:
Alice encrypts Va using Paillier, resulting in Ea = [E(a1), E(a2), ..., E(a10)].
Bob encrypts Vb using Paillier, resulting in Eb = [E(b1), E(b2), ..., E(b10)].
Chris encrypts Vc using Paillier, resulting in Ec = [E(c1), E(c2), ..., E(c10)].
David encrypts Vd using Paillier, resulting in Ed = [E(d1), E(d2), ..., E(d10)].

Key code:

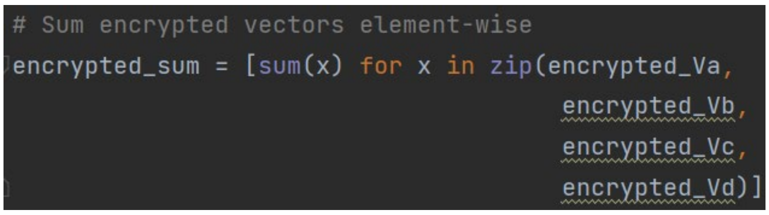


```
# Generate Paillier keypair
public_key, private_key = paillier.generate_paillier_keypair()
```



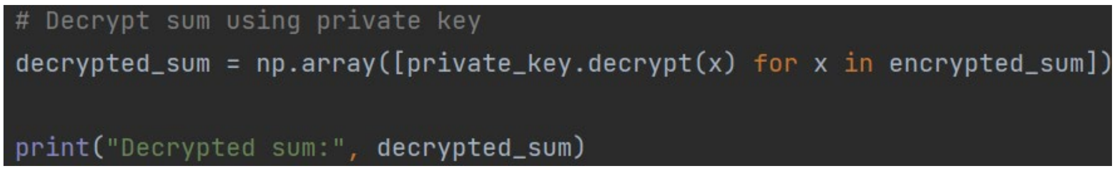
```
# Encrypt vectors using public key
encrypted_Va = [public_key.encrypt(x) for x in Va]
encrypted_Vb = [public_key.encrypt(x) for x in Vb]
encrypted_Vc = [public_key.encrypt(x) for x in Vc]
encrypted_Vd = [public_key.encrypt(x) for x in Vd]
```

Computes the sum of the encrypted vectors using homomorphic addition.
Esum = [E(a1 + b1 + c1 + d1), E(a2 + b2 + c2 + d2), ..., E(a10 + b10 + c10 + d10)].



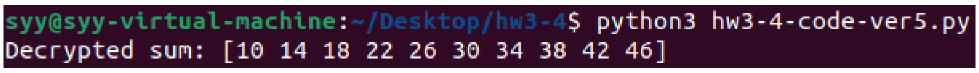
```
# Sum encrypted vectors element-wise
encrypted_sum = [sum(x) for x in zip(encrypted_Va,
                                     encrypted_Vb,
                                     encrypted_Vc,
                                     encrypted_Vd)]
```

Step3. Decrypt the resulting ciphertexts to obtain the plain sum vector, as follows:
sum_plain = [a1 + b1 + c1 + d1, a2 + b2 + c2 + d2, ..., a10 + b10 + c10 + d10].



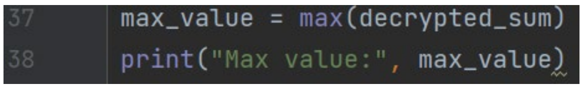
```
# Decrypt sum using private key
decrypted_sum = np.array([private_key.decrypt(x) for x in encrypted_sum])

print("Decrypted sum:", decrypted_sum)
```

Here is the output of sum after decryption:



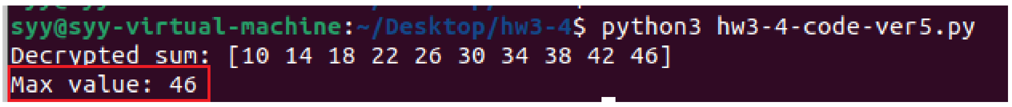
```
syy@syy-virtual-machine:~/Desktop/hw3-4$ python3 hw3-4-code-ver5.py
Decrypted sum: [10 14 18 22 26 30 34 38 42 46]
```

Step5. Finds the maximum value in the plain sum vector via max() function:
max_value = max(sum_plain).



```
max_value = max(decrypted_sum)
print("Max value:", max_value)
```

Here is the output:



```
syy@syy-virtual-machine:~/Desktop/hw3-4$ python3 hw3-4-code-ver5.py
Decrypted sum: [10 14 18 22 26 30 34 38 42 46]
Max value: 46
```

Test:

Regarding the vectors I defined:
# Alice's vector
Va = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
# Bob's vector
Vb = np.array([2, 3 ,4 ,5 ,6 ,7 ,8 ,9 ,10 ,11])
# Chris' vector
Vc = np.array([3 ,4 ,5 ,6 ,7 ,8 ,9 ,10 ,11 ,12])
# David's vector
Vd = np.array([4 ,5 ,6 ,7 ,8 ,9 ,10 ,11,12,13])
The last element is the maximum for each vector, hence: $Vsum_{max}$ = 10+11+12+13=46
$Vsum_{max}$=46 is the same as the max value we get from the code implementation.

Way 2 Use a secure multi-party computation (MPC) protocol based on garbled circuits (Fairplay)

My second protocol design uses a secure multi-party computation (MPC) protocol based on garbled circuits to sum the four vectors and find the maximum value in V. This can be achieved using a garbled circuit (Fairplay). In this protocol, each party would input their respective vector into the garbled circuit. Here are the key steps:
Step 1: Alice, Bob, Chris and David each input their respective vector (10 integers in each vector) into the garbled circuit. Define and set 4 parties:
For Alice: Va = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
For Bob: Vb = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
For Chris: Vc = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
For David: Vd = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Obviously, Vmax = 10+10+10+10 =40. Let's implement how to get the max sum V without disclosing to any party in the following steps.

Step 2: The garbled circuit I use takes as input the four vectors Va, Vb, Vc, and Vd and outputs their sum V based on the index of the maximum value in V. The circuit will consist of two parts:

1) The first part computes the sum of the vectors, element-wise.
2) The second part finds the maximum value in the sum via a comparison of iterating based on the index.

Here is the key code (Detailed implementation is shown in Task2):

```
for (i = 0 to inp_size - 1) {
   current_sum = input.alice[i] + input.bob[i] + input.chris[i] + input.david[i];
   if (current_sum > max_sum) {
     max_sum = current_sum;
     }
}

output.alice = max_sum;
output.bob = max_sum;
output.chris= max_sum;
output.david= max_sum;
```

We can garble it using a garbling algorithm such as Yao's garbling algorithm in this step. The garbled circuit will be sent to all parties. Fairplay is implemented based on Yao's algorithm. The algorithm works as follows:

1. The party with the circuit (the "garbler") assigns random values to each wire in the circuit.
2. The garbler then constructs a "garbled table" for each gate in the circuit. A garbled table is a table of encrypted values that represent the output of each possible combination of inputs to the gate.
3. The garbler sends the garbled circuit and the garbled tables to the second party (the "evaluator").
4. The evaluator uses an oblivious transfer protocol to select the appropriate row from each garbled table, based on the input values provided by the garbler. This allows the evaluator to determine the output of each gate in the circuit without knowing the underlying values of the inputs or outputs.
5. Finally, the evaluator uses the outputs from each gate to determine the overall output of the circuit.

Step 3: Each party will mask their input vector by adding two vectors of the same size twice. Specifically, each party Pi will compute a masked input vector Vi' = Vi + Ri, where Ri is a generated vector of the same size as Vi. The circuit computes the sum of every two vectors which are sorted separately, then finds the maximum value by comparison in V securely.

Task 2 Code Implementation

Here is the code implementation of Way 1 which is using Homomorphic Encryption (Paillier's Cryptosys- tem) + Secure Multiparty Computation

hw3-4-code-HE.py.

When you run the script which is python code, make sure your system has installed phe, numpy libriries

Here is the explanation of the code explanation based on Way 2 which is using a secure multi-party computation (MPC) protocol based on garbled circuits.

The scripts are:

(1) Files for compiling:

hw3-4-MaxAliceBob.txt
hw3-4-MaxChrisDavid.txt

(2) Command:

hw3-4-run_chris
hw3-4-run_david

The outputs are garbled circuits we use int the protocol:

hw3-4-MaxAliceBob.txt.Opt.circuit
hw3-4-MaxAliceBob.txt.Opt.fmt
hw3-4-MaxChrisDavid.txt.Opt.circuit
hw3-4-MaxChrisDavid.txt.Opt.fmt

Readme files:

hw3-4-Readme-MaxBobAlice.txt
hw3-4-Readme-MaxChrisDavid.txt

Note:
Run the script based on the readme files. You might need to edit the name of the scripts. Because we are required to upload all documents in certain Naming Conventions, I have to follow the requirements and add the prefix e.g. 'hw3-3'.

Step 1. Given the for vectors stand for the four parties are the same respectively (10 integers in each vector):

For Alice: Va = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
For Bob: Vb = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
For Chris: Vc = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
For David: Vd = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

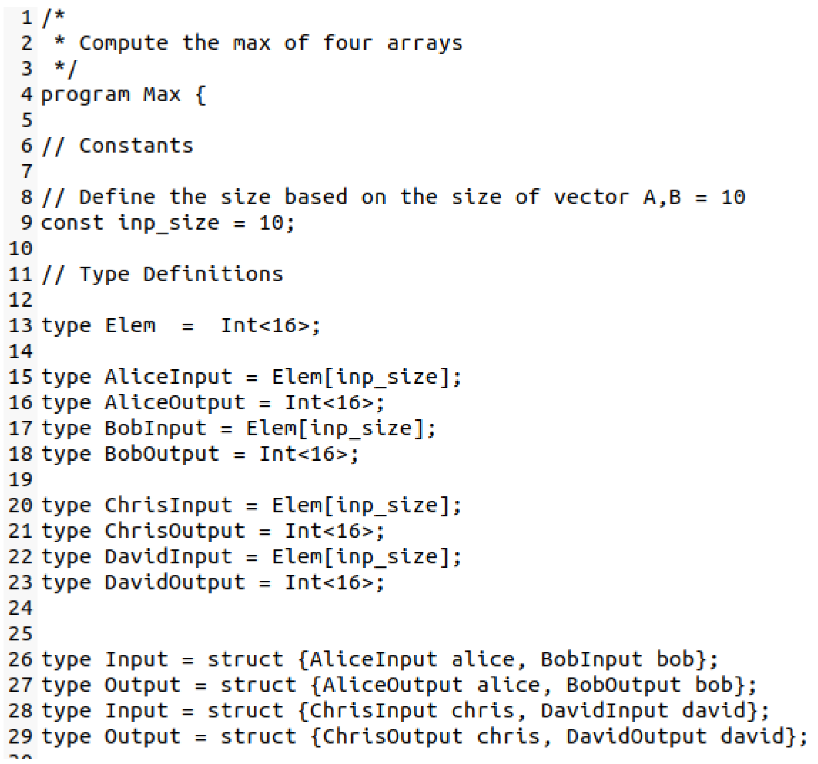
Note: index starts from 0. Apparently, sum all the four vectors: V = Va + Vb + Vc + Vd, and find the maximum value (out of the 10 entries) in V is Vmax = Max(Va + Vb + Vc + Vd) = 40. Let's implement it based on the requirements of Problem 4.

Step 2. Design and implement SFDL program for finding the Vmax

The max SFDL program is designed based on the below program structure:

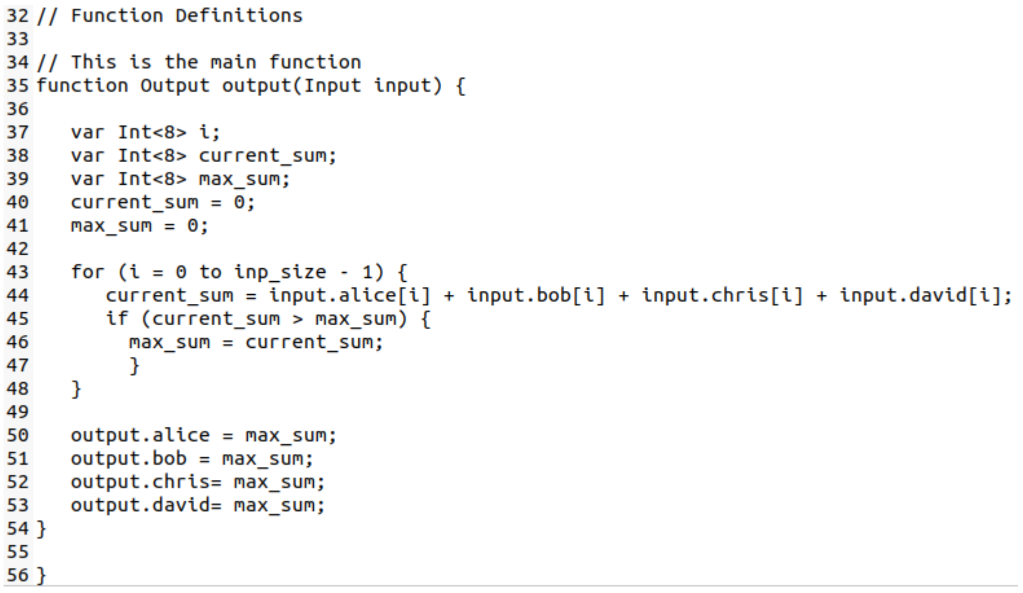
```
program <program-name> {
   <type declarations>
   <function declarations>
}
```

Here is the type I defined:



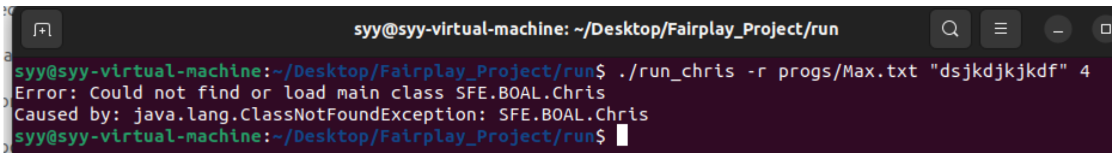
```
/*
 * Compute the max of four arrays
 */
program Max {

// Constants

// Define the size based on the size of vector A,B = 10
const inp_size = 10;

// Type Definitions

type Elem  =  Int<16>;

type AliceInput = Elem[inp_size];
type AliceOutput = Int<16>;
type BobInput = Elem[inp_size];
type BobOutput = Int<16>;

type ChrisInput = Elem[inp_size];
type ChrisOutput = Int<16>;
type DavidInput = Elem[inp_size];
type DavidOutput = Int<16>;


type Input = struct {AliceInput alice, BobInput bob};
type Output = struct {AliceOutput alice, BobOutput bob};
type Input = struct {ChrisInput chris, DavidInput david};
type Output = struct {ChrisOutput chris, DavidOutput david};
```

Here is the function for getting the maximum of sum I defined:



```
// Function Definitions

// This is the main function
function Output output(Input input) {

   var Int<8> i;
   var Int<8> current_sum;
   var Int<8> max_sum;
   current_sum = 0;
   max_sum = 0;

   for (i = 0 to inp_size - 1) {
      current_sum = input.alice[i] + input.bob[i] + input.chris[i] + input.david[i];
      if (current_sum > max_sum) {
        max_sum = current_sum;
        }
   }

   output.alice = max_sum;
   output.bob = max_sum;
   output.chris= max_sum;
   output.david= max_sum;
}

}
```

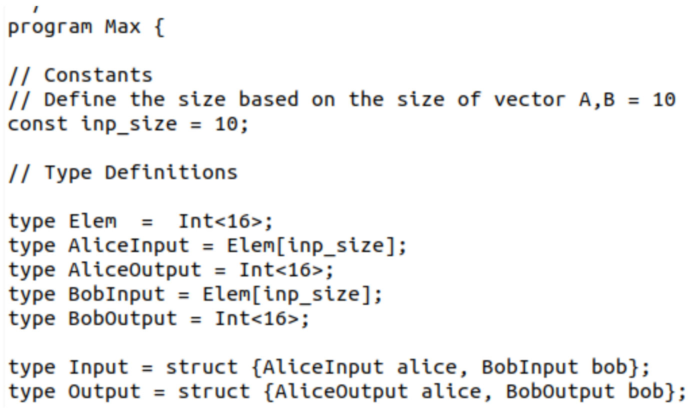
Compile it for Alice and Bob, Chris, David and run the protocol (communication is integrated in Fairplay). When I run the SFDL code, I found the problem which is needed us to edit the black box which is the jar file we applied:



```
syy@syy-virtual-machine:~/Desktop/Fairplay_Project/run$ ./run_chris -r progs/Max.txt "dsjkdjkjkdf" 4
Error: Could not find or load main class SFE.BOAL.Chris
Caused by: java.lang.ClassNotFoundException: SFE.BOAL.Chris
syy@syy-virtual-machine:~/Desktop/Fairplay_Project/run$
```

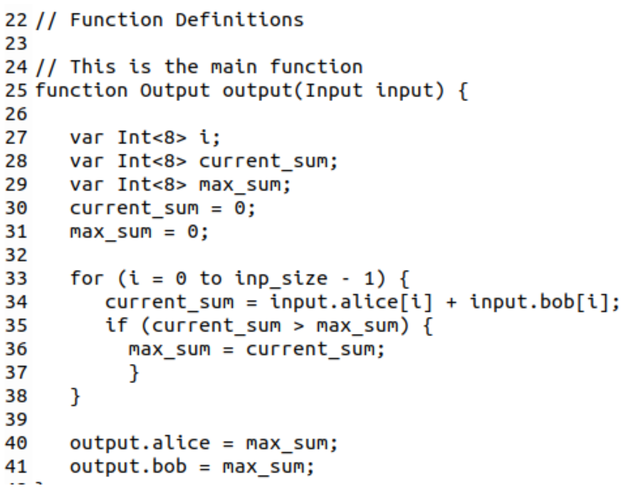
Regarding 'jars' files we have to use in 'Fairplay_Project', here I will implement the code based on sorted arrays to implement the encryption and decryption process based on two parties first. Hence, the maximum sum = Vmax = Max(Va+Vb) + Max(Vc+Vd)

Here is the type I defined for Alice and Bob:



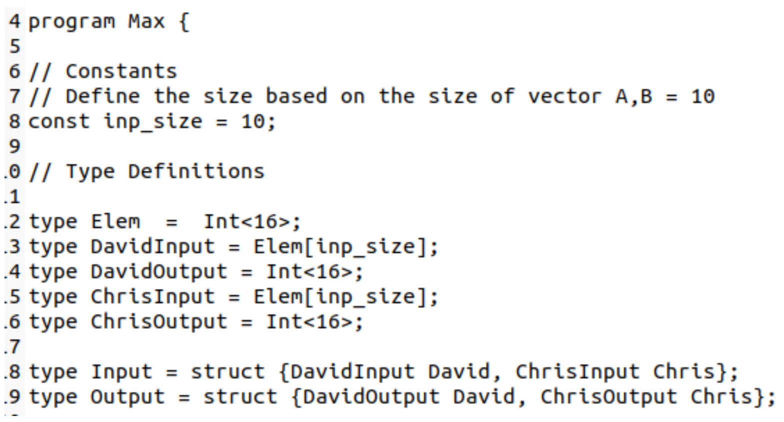
```
program Max {

// Constants
// Define the size based on the size of vector A,B = 10
const inp_size = 10;

// Type Definitions

type Elem  =  Int<16>;
type AliceInput = Elem[inp_size];
type AliceOutput = Int<16>;
type BobInput = Elem[inp_size];
type BobOutput = Int<16>;

type Input = struct {AliceInput alice, BobInput bob};
type Output = struct {AliceOutput alice, BobOutput bob};
```

Here is the function I defined for Alice and Bob:



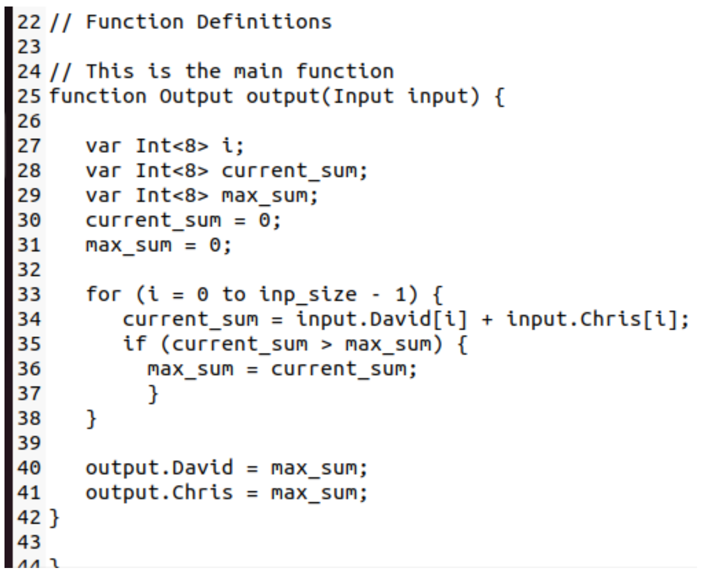
```
// Function Definitions

// This is the main function
function Output output(Input input) {

   var Int<8> i;
   var Int<8> current_sum;
   var Int<8> max_sum;
   current_sum = 0;
   max_sum = 0;

   for (i = 0 to inp_size - 1) {
      current_sum = input.alice[i] + input.bob[i];
      if (current_sum > max_sum) {
        max_sum = current_sum;
        }
   }

   output.alice = max_sum;
   output.bob = max_sum;
```

Here is the type I defined for Chris and David:



```
program Max {

// Constants
// Define the size based on the size of vector A,B = 10
const inp_size = 10;

// Type Definitions

type Elem  =  Int<16>;
type DavidInput = Elem[inp_size];
type DavidOutput = Int<16>;
type ChrisInput = Elem[inp_size];
type ChrisOutput = Int<16>;

type Input = struct {DavidInput David, ChrisInput Chris};
type Output = struct {DavidOutput David, ChrisOutput Chris};
```

Here is the function I defined for Chris and David:



```
// Function Definitions

// This is the main function
function Output output(Input input) {

   var Int<8> i;
   var Int<8> current_sum;
   var Int<8> max_sum;
   current_sum = 0;
   max_sum = 0;

   for (i = 0 to inp_size - 1) {
      current_sum = input.David[i] + input.Chris[i];
      if (current_sum > max_sum) {
        max_sum = current_sum;
        }
   }

   output.David = max_sum;
   output.Chris = max_sum;
}

}
```

Write the command for Chris and David:

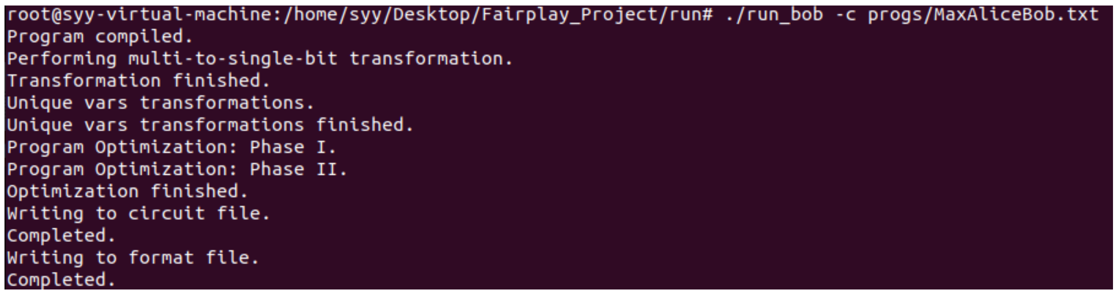
hw3-4-run_chris

hw3-4-run_david

Step 2. Compile it for Alice, Bob, Chris, David and run the protocol (communication is integrated in Fairplay).

(1) Compile for the party who is called Bob:

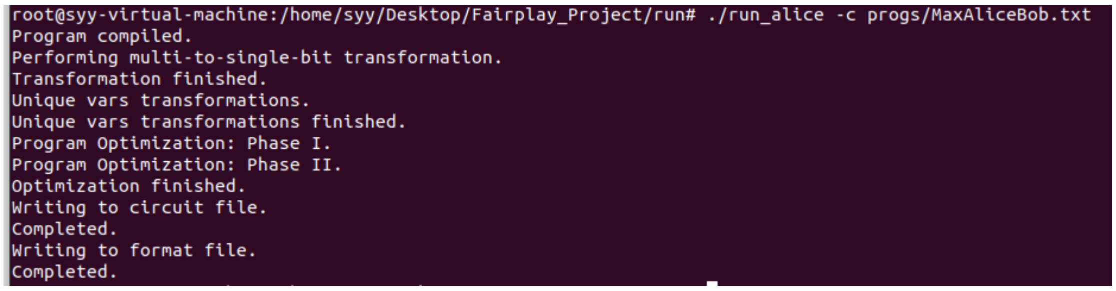
./run_bob -c progs/MaxAliceBob.txt



```
root@syy-virtual-machine:/home/syy/Desktop/Fairplay_Project/run# ./run_bob -c progs/MaxAliceBob.txt
Program compiled.
Performing multi-to-single-bit transformation.
Transformation finished.
Unique vars transformations.
Unique vars transformations finished.
Program Optimization: Phase I.
Program Optimization: Phase II.
Optimization finished.
Writing to circuit file.
Completed.
Writing to format file.
Completed.
```

(2) Compile for the party who is called Alice:

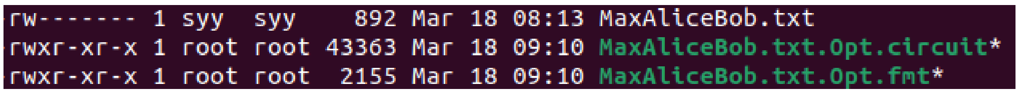
./run_alice -c progs/MaxAliceBob.txt



```
root@syy-virtual-machine:/home/syy/Desktop/Fairplay_Project/run# ./run_alice -c progs/MaxAliceBob.txt
Program compiled.
Performing multi-to-single-bit transformation.
Transformation finished.
Unique vars transformations.
Unique vars transformations finished.
Program Optimization: Phase I.
Program Optimization: Phase II.
Optimization finished.
Writing to circuit file.
Completed.
Writing to format file.
Completed.
```

After compiling Alice and Bob, we see all commands produce the same two output files:

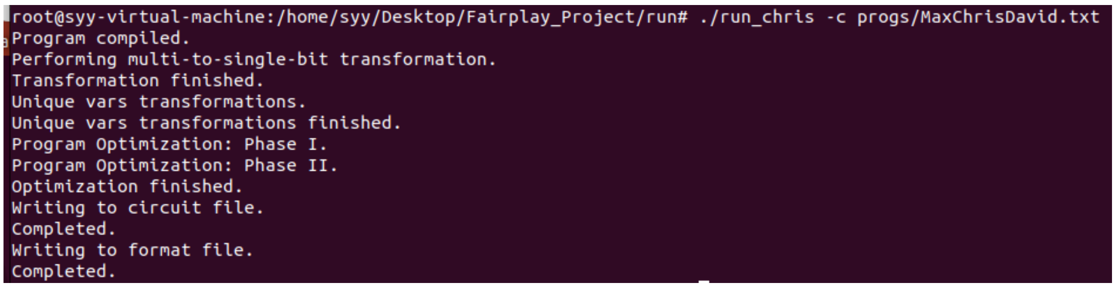
1. progs/MaxAliceBob.txt.Opt.circuit (an SHDL circuit)
2. progs/MaxAliceBob.txt.Opt.fmt



```
rw------- 1 syy  syy    892 Mar 18 08:13 MaxAliceBob.txt
rwxr-xr-x 1 root root 43363 Mar 18 09:10 MaxAliceBob.txt.Opt.circuit*
rwxr-xr-x 1 root root  2155 Mar 18 09:10 MaxAliceBob.txt.Opt.fmt*
```

(3) Compile for the party who is called Chris:

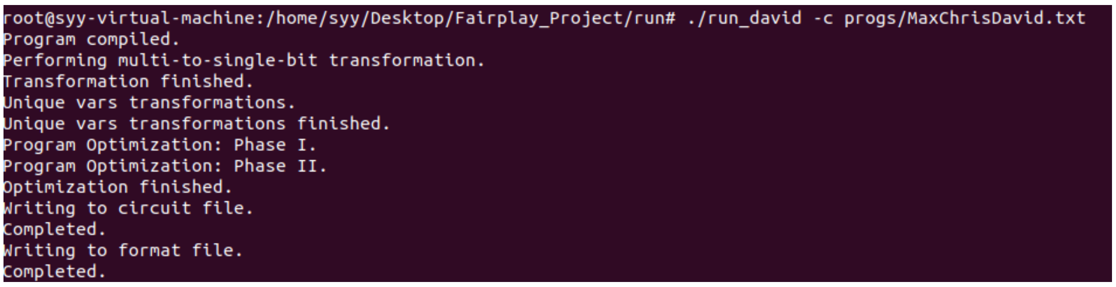
./run_chris -c progs/MaxChrisDavid.txt



```
root@syy-virtual-machine:/home/syy/Desktop/Fairplay_Project/run# ./run_chris -c progs/MaxChrisDavid.txt
Program compiled.
Performing multi-to-single-bit transformation.
Transformation finished.
Unique vars transformations.
Unique vars transformations finished.
Program Optimization: Phase I.
Program Optimization: Phase II.
Optimization finished.
Writing to circuit file.
Completed.
Writing to format file.
Completed.
```

(4) Compile for the party who is called David:

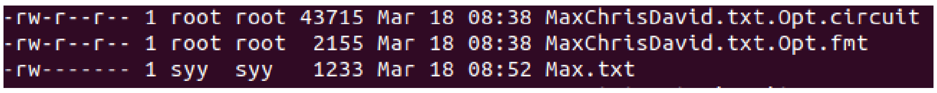
./run_david -c progs/MaxChrisDavid.txt



```
root@syy-virtual-machine:/home/syy/Desktop/Fairplay_Project/run# ./run_david -c progs/MaxChrisDavid.txt
Program compiled.
Performing multi-to-single-bit transformation.
Transformation finished.
Unique vars transformations.
Unique vars transformations finished.
Program Optimization: Phase I.
Program Optimization: Phase II.
Optimization finished.
Writing to circuit file.
Completed.
Writing to format file.
Completed.
```

After compiling Chris and David, we see all commands produce the same two output files:

1. progs/MaxChrisDavid.txt.Opt.circuit (an SHDL circuit)
2. progs/ axChrisDavid.txt.Opt.fmt



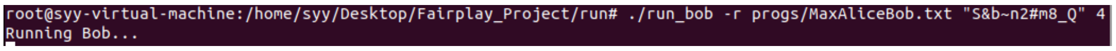
```
-rw-r--r-- 1 root root 43715 Mar 18 08:38 MaxChrisDavid.txt.Opt.circuit
-rw-r--r-- 1 root root  2155 Mar 18 08:38 MaxChrisDavid.txt.Opt.fmt
-rw------- 1 syy  syy   1233 Mar 18 08:52 Max.txt
```

Step 3. Run the protocol

Communication is integrated in Fairplay. According to theory and explanation in the paper (Fairplay — A Secure Two-Party Computation System), it's considered several flavors of oblivious transfer(OT) algorithms in this Fairplay tool. Since 4 is the best one among oblivious transfer(OT) algorithms, I will use OT type = 4 to run the protocol.

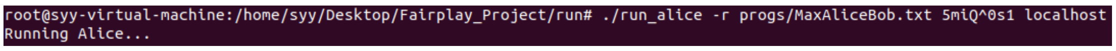
Run Bob (should be first):

./run_bob -r progs/MaxAliceBob.txt "S&b~n2#m8_Q" 4



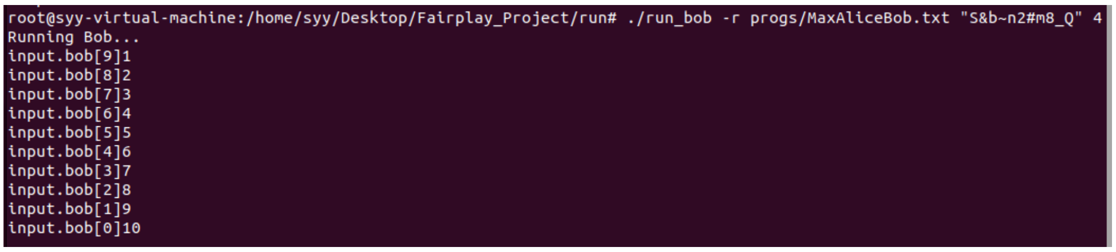
```
root@syy-virtual-machine:/home/syy/Desktop/Fairplay_Project/run# ./run_bob -r progs/MaxAliceBob.txt "S&b~n2#m8_Q" 4
Running Bob...
```

./run_alice -r progs/MaxAliceBob.txt 5miQ^0s1 localhost

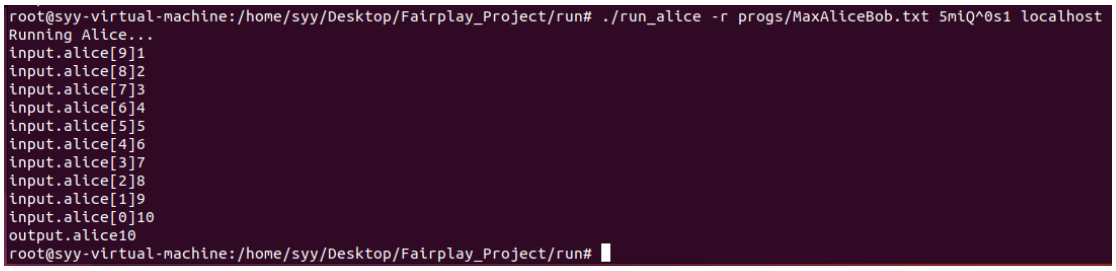


```
root@syy-virtual-machine:/home/syy/Desktop/Fairplay_Project/run# ./run_alice -r progs/MaxAliceBob.txt 5miQ^0s1 localhost
Running Alice...
```

Input elements for Vectors Va, Vb:

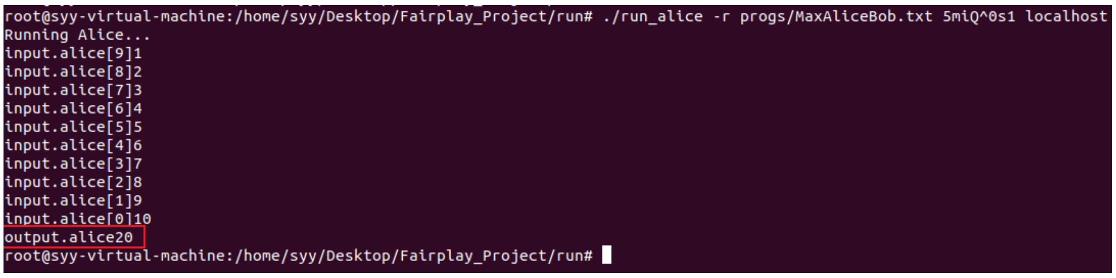


```
root@syy-virtual-machine:/home/syy/Desktop/Fairplay_Project/run# ./run_bob -r progs/MaxAliceBob.txt "S&b~n2#m8_Q" 4
Running Bob...
input.bob[9]1
input.bob[8]2
input.bob[7]3
input.bob[6]4
input.bob[5]5
input.bob[4]6
input.bob[3]7
input.bob[2]8
input.bob[1]9
input.bob[0]10
```

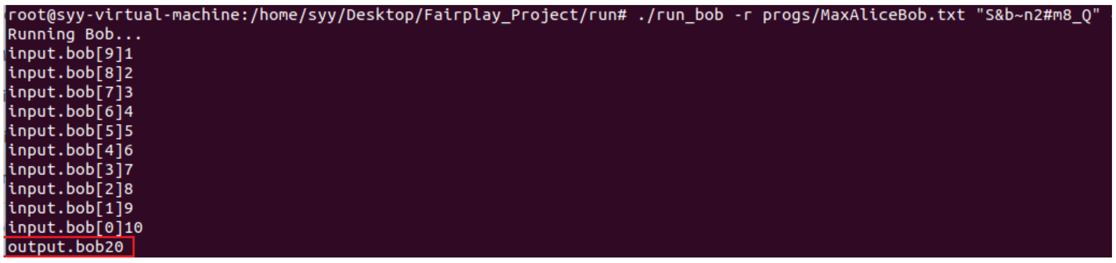


```
root@syy-virtual-machine:/home/syy/Desktop/Fairplay_Project/run# ./run_alice -r progs/MaxAliceBob.txt 5miQ^0s1 localhost
Running Alice...
input.alice[9]1
input.alice[8]2
input.alice[7]3
input.alice[6]4
input.alice[5]5
input.alice[4]6
input.alice[3]7
input.alice[2]8
input.alice[1]9
input.alice[0]10
output.alice10
root@syy-virtual-machine:/home/syy/Desktop/Fairplay_Project/run#
```

Here is the output: We get the maximum sum = Max(Va + Vb) = 20



```
root@syy-virtual-machine:/home/syy/Desktop/Fairplay_Project/run# ./run_alice -r progs/MaxAliceBob.txt 5miQ^0s1 localhost
Running Alice...
input.alice[9]1
input.alice[8]2
input.alice[7]3
input.alice[6]4
input.alice[5]5
input.alice[4]6
input.alice[3]7
input.alice[2]8
input.alice[1]9
input.alice[0]10
output.alice20
root@syy-virtual-machine:/home/syy/Desktop/Fairplay_Project/run#
```

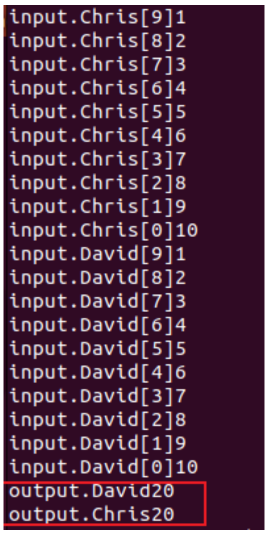


```
root@syy-virtual-machine:/home/syy/Desktop/Fairplay_Project/run# ./run_bob -r progs/MaxAliceBob.txt "S&b~n2#m8_Q"
Running Bob...
input.bob[9]1
input.bob[8]2
input.bob[7]3
input.bob[6]4
input.bob[5]5
input.bob[4]6
input.bob[3]7
input.bob[2]8
input.bob[1]9
input.bob[0]10
output.bob20
```

./run_chris -r progs/MaxChrisDavid.txt "S&b~n2#m8_Q" 4

./run_david -r progs/MaxChrisDavid.txt 5miQ^0s1 localhost

Input elements for Vectors Vc, Vd we defined previously, we get the maximum sum = Max(Vc + Vd) = 20



```
input.Chris[9]1
input.Chris[8]2
input.Chris[7]3
input.Chris[6]4
input.Chris[5]5
input.Chris[4]6
input.Chris[3]7
input.Chris[2]8
input.Chris[1]9
input.Chris[0]10
input.David[9]1
input.David[8]2
input.David[7]3
input.David[6]4
input.David[5]5
input.David[4]6
input.David[3]7
input.David[2]8
input.David[1]9
input.David[0]10
output.David20
output.Chris20
```

Hence, the SDFL program securely sums all the four vectors, finding the maximum value (out of the 10 entries) in V that is Vmax = Max(Va+Vb+Vc+Vd) = 20+20=40. Sum V is not disclosed to any party.