Problem 3 Private Information Retrieval

In the class, we talked about using homomorphic encryption to perform private information retrieval (PIR). Specifically, suppose the server holds a dataset D of n integers d = [d1, d2, · · · , dn] encrypted under the client's key. For the server to select an element, e.g., di, obliviously, the client sends an encrypted selection binary vector s of length n containing 0s everywhere except for a 1 in the position i of the element to be selected. The server then computes the dot product of d and s using the homomorphism, with the encrypted result $r = \sum_{j=1}^{n} d_j s_j$, and sends it to the client, who can decrypt it.

1. Protocol design. Please design a protocol based on (somewhat/fully) homomorphic encryption to achieve the goal.

My solution:

Here is my design based on homomorphic encryption to achieve the goal of private information retrieval (PIR):

Assumptions: The client has a public key and a secret key, and the server has the encrypted dataset D. Both client and server have access to a homomorphic encryption library, such as HElib, SEAL, or TFHE.

1. The client generates an encryption key pair and sends the public key to the server. Concretely, the client generates n random **binary values s = [s1, s2, ..., sn]**:
   - **n:** the size of vector s that the client holds should be the same as the size of the vector the sever holds.
   - **vector s** that the client holds containing 0s everywhere except for a 1 in the position i of the element to be selected

   The client encrypts each value using homomorphic encryption, and sends the encrypted vector [Enc(s1), Enc(s2), ..., Enc(sn)] to the server. For example,

   The client sends: $\begin{bmatrix} E(0) \\ E(0) \\ E(1) \end{bmatrix}$

2. The server encrypts the dataset D under the client's public key and sends it back to the client. Concretely, The server performs the homomorphic dot product of the encrypted vector [Enc(d1), Enc(d2), ..., Enc(dn)] and the encrypted vector [Enc(s1), Enc(s2), ..., Enc(sn)], resulting in the encrypted result Enc(r).

   For example,

The server then compute:

$$\begin{bmatrix} D_1 \cdot E(0) + D_2 \cdot E(0) + D_3 \cdot E(1) \\ D_4 \cdot E(0) + D_5 \cdot E(0) + D_6 \cdot E(1) \\ D_7 \cdot E(0) + D_8 \cdot E(0) + D_9 \cdot E(1) \end{bmatrix} = \begin{bmatrix} E(D_3) \\ E(D_6) \\ E(D_9) \end{bmatrix}$$

3. The server sends the encrypted result Enc(r) to the client. The client decrypts the result using its secret key to obtain the selected element di = r.

4. The client repeats step 3 for each element it wants to retrieve (The client receives this value, decrypts, and gets the result).

Note:
This protocol I designed above can ensure that the server cannot learn which element the client is requesting since the client sends an encrypted binary vector s that only reveals the position of the requested element. The server can only compute the dot product of the encrypted dataset and the encrypted selection vector and send back the encrypted result, which the client can decrypt to obtain the selected element without revealing any information to the server.

2. Implementation.

This simple dot-product protocol for a database of a specific size can be run by the the SHEEP platform: https://github.com/alan-turing-institute/SHEEP.

SHEEP supports a number of fully homomorphic encryption libraries (including HElib, SEAL TFHE, and libpallier). Please use the SHEEP platform to realize the PIR, where the server holds d = [2, 4, 6, 8, 10, 1, 3] and the client wants to retrieve each element.

Submission requirement: (1) screenshots of the major steps of your answer, and (2) source code files – all named with the prefix “hw4-3-”.

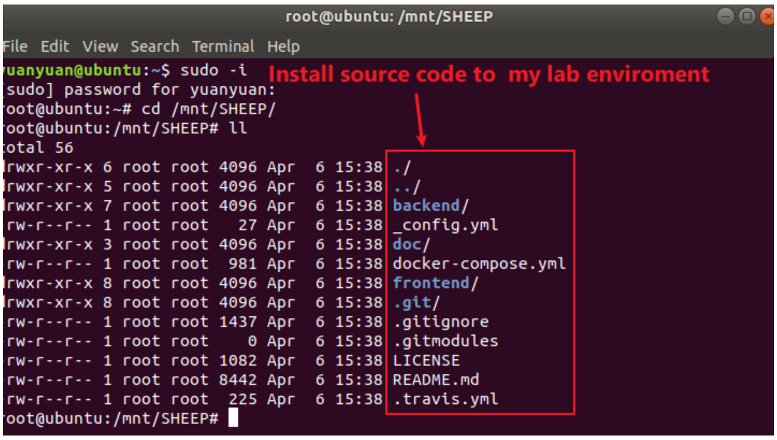
My solution:

Lab Environment:

- Ubuntu 18.04
- Docker (version 23.0.3)
- Docker compose
- Source code of SHEEP
- Python 3

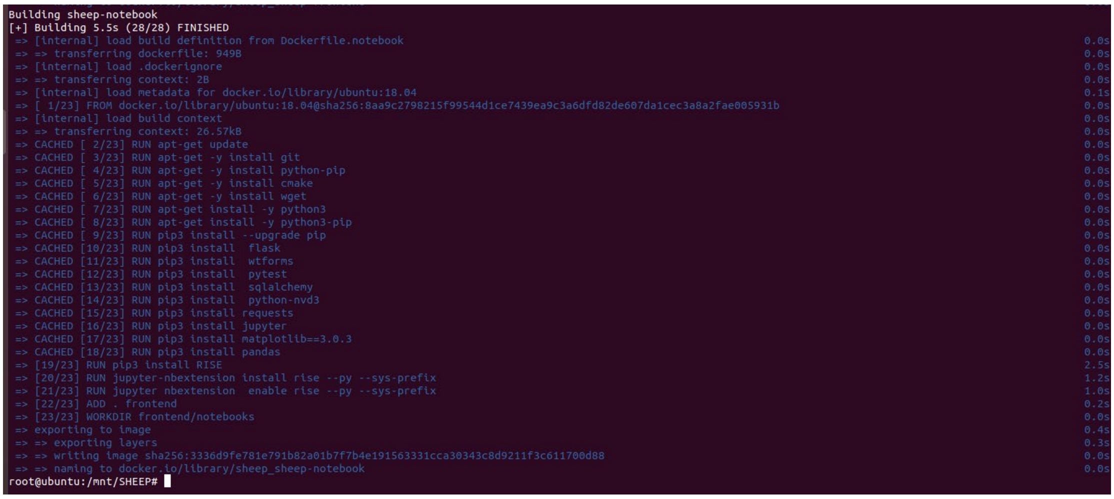
Step I. Install Source Code

Source code url: https://github.com/alan-turing-institute/SHEEP
Command to install code: git clone https://github.com/alan-turing-institute/SHEEP.git



Step II. Build Environment for SHEEP

Go to the directory that the source code installed that has “docker-compose.yml” and run the command: docker-compose build



Now, there’s no error information. The environment for SHEEP is built successfully.

Note: This step took me more than 10 minutes because there are many libraries that needed to be installed. I work on VM 15 with Ubuntu, and my laptop screen froze many times.

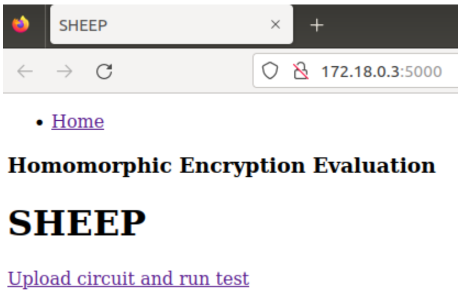
Step III. Deploy Source Code via Docker (Deploy sheep-frontend)

Note: in this step, I simply introduce SHEEP and how we can use SHEEP to implement private information retrieval.

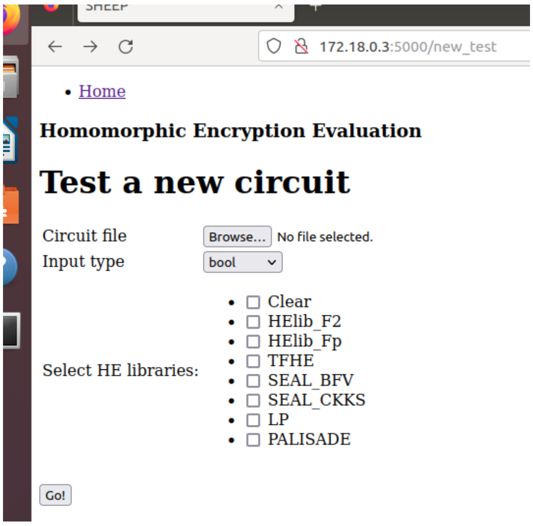
Enter the directory the source code installed that has “docker-compose.yml” and run the command: docker-compose run -p 5000:5000 sheep-frontend

```
root@ubuntu:/mnt/SHEEP# docker-compose run -p 5000:5000 sheep-frontend
Creating network "sheep_sheep_nw" with driver "bridge"
Creating sheep_sheep-server_1 ... done
Creating sheep_sheep-frontend_run ... done
 * Serving Flask app 'app' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://172.18.0.3:5000/ (Press CTRL+C to quit)
```

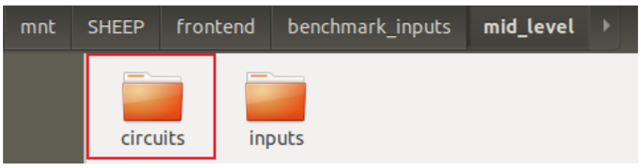
Now, the SHEEP server is created and deployed successfully. The SHEEP web interface (frontend) is running.

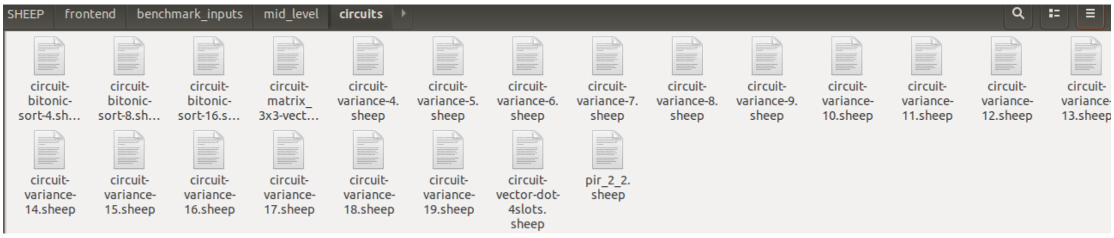


The SHEEP project is to provide a platform for practitioners to evaluate the state-of-the-art of (fully) homomorphic encryption technology in the context of their concrete application. SHEEP enables this evaluation to be done across libraries implementing different HE schemes (with in some cases incomparable security). The libraries we can use shown in the picture below are HElib, TFHE, LP, etc.



The circuits we could use are provided via the source code. Here are the circuits we can apply. For this question, we need to use the circuit in mid_level.





There are two ways we can use SHEEP to implement private information retrieval.

Way 1

We can use the dot product script in jupyter notebook to design the dot circuit with 7 slots via using ROTATE and ADD operations to implement private information retrieval.

Way 2

We can try to design the dot-product script in SHEEP language and perform it on SHEEP web interface. I will explain a little bit in the last part.

Step IV. Realize PIR

Way 1 Work on Jupyter Notebook

Let's use the dot product script in jupyter notebook to design the dot circuit with 7 slots via using ROTATE and ADD operations to implement private information retrieval.

A powerful feature of some HE schemes is the ability to perform SIMD operations, doing the same calculation on multiple "slots" (i.e. elements of a vector). The first part of a vector dot product - the component-wise multiplication - is therefore trivial. However, we then need to sum over the elements to obtain the scalar product. This can be done using ROTATE and ADD operations, as demonstrated in this notebook.

Here is the input vectors:

intput_0: The vector d that the server holds which is d = [2, 4, 6, 8, 10, 1, 3]

intput_1: The vector s that the client sends. The vector s contains 0s everywhere except for a 1 in the position i of the element to be selected

Note: It means the client needs to send the vector s with a different index where the value is 1 for 7 times such that the client can retrieve all the elements in the vector the server holds.

1^{st} time: [2, 4, 6, 8, 10, 1, 3] X [1, 0, 0, 0, 0, 0, 0] = **2**+0+0+0+0+0+0 = 2

2^{nd} time: [2, 4, 6, 8, 10, 1, 3] X [0, 1, 0, 0, 0, 0, 0] = 0+**4**+0+0+0+0+0 = 4

3^{rd} time: [2, 4, 6, 8, 10, 1, 3] X [0, 0, 1, 0, 0, 0, 0] = 0+0+**6**+0+0+0+0 = 6

4^{th} time: [2, 4, 6, 8, 10, 1, 3] X [0, 0, 0, 1, 0, 0, 0] = 2+0+0+**8**+0+0+0 = 8

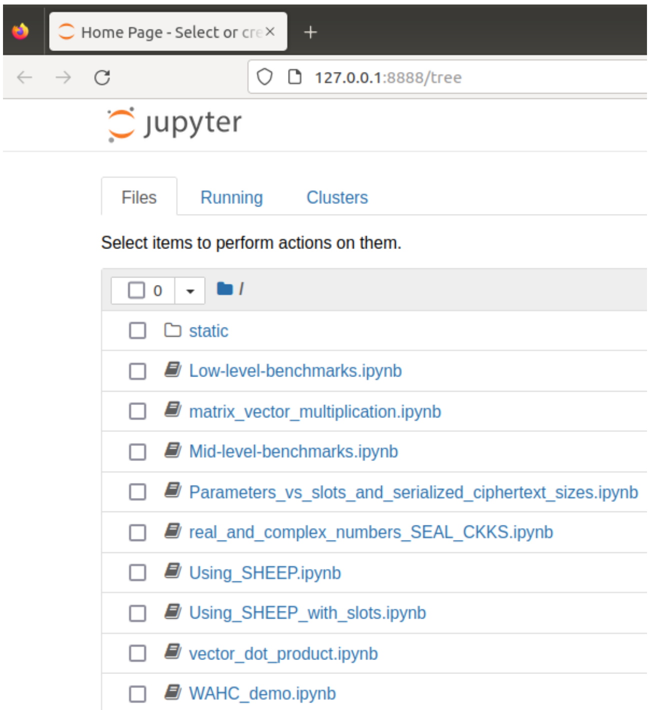
5^{th} time: [2, 4, 6, 8, 10, 1, 3] X [0, 0, 0, 0, 1, 0, 0] = 0+0+0+0+**10**+0+0 = 10

6^{th} time: [2, 4, 6, 8, 10, 1, 3] X [0, 0, 0, 0, 0, 1, 0] = 0+0+0+0+0+**1**+0 = 1

7^{th} time: [2, 4, 6, 8, 10, 1, 3] X [0, 0, 0, 0, 0, 0, 1] = 0+0+0+0+0+0+**3** = 3

Step 1 Run the command: docker-compose run -p 8888:8888 sheep-notebook to work on Jupyter Notebook.

```
root@ubuntu:/mnt/SHEEP# docker-compose run -p 8888:8888 sheep-notebook
Creating sheep_sheep-notebook_run ... done
[I 02:36:31.591 NotebookApp] Writing notebook server cookie secret to /root/.local/share/jupyter/runtime/notebook_cookie_secret
[I 02:36:32.183 NotebookApp] Serving notebooks from local directory: /frontend/notebooks
[I 02:36:32.183 NotebookApp] Jupyter Notebook 6.4.10 is running at:
[I 02:36:32.184 NotebookApp] http://b730b52941ed:8888/?token=063732985fd52059e071f496f2aa6f6992ed908f62a7fbcd
[I 02:36:32.184 NotebookApp]  or http://127.0.0.1:8888/?token=063732985fd52059e071f496f2aa6f6992ed908f62a7fbcd
```



Note: we need to work on vector_dot_product.ipynb.

Step2. Load libraries for the python code

```
In [1]: import os
        if "SHEEP_HOME" in os.environ.keys():
          SHEEP_HOME = os.environ["SHEEP_HOME"]
        else:
          print("Please set environment variable SHEEP_HOME to point to location of SHEEP/frontend")
        import sys
        sys.path.append(SHEEP_HOME)

        from pysheep import sheep_client
```

Step3. Start a new job

```
In [2]: '''
        Step3. Start a new job

        Yuanyuan Sun comment:
        The step ID here is based on my operation/experiment. Whenever you need to start a new job,
        you need to perform this step.
        '''
        sheep_client.new_job()

Out[2]: {'status_code': 200, 'content': ''}
```

We see the new job is started successfully.

Step 4. Use the HE(homomorphic encryption) scheme which is SEAL_BFV

In [30]:
```
'''
Step 4. Use the HE(homomorphic encryption) scheme which is SEAL_BFV
'''
sheep_client.set_context("SEAL_BFV")
sheep_client.set_input_type("int8_t")
sheep_client.get_nslots()
```

Out[30]: {'status_code': 200, 'content': {'nslots': 2048}}

Step 5. Generalizing, and generating circuits

Here is the key step: using ROTATE and ADD operations to obtain the scalar product

In [31]:
```
def generate_vector_dot_product_circuit(input_0, input_1):
    """
    Given two input lists (must be equal in length) generate a SHEEP circuit to do the dot product
    """
    if len(input_0) != len(input_1):
        raise RuntimeError("input_0 and input_1 must be the same length")
    circuit_str = "INPUTS input_0 input_1\nCONST_INPUTS rotate_1\nOUTPUTS output prod_s1 prod_s2 prod_s3\ninput_0 inp
    for i in range(len(input_0)-1):
        circuit_str += "prod_r{} rotate_1 ROTATE prod_r{}\n".format(i,i+1)
        if i==0:
            circuit_str += "prod_r0 prod_r1 ADD prod_s1\n"
        else:
            circuit_str += "prod_s{} prod_r{} ADD prod_s{}\n".format(i,i+1,i+1)
    circuit_str += "prod_s{} ALIAS output\n".format(i+1)
    return circuit_str
```

Step 6. Calculate Dot Product

So, let's do the calculation in SEAL_BFV - multiply 2 vectors with 7 elements each, where each element.

Step 6.1 Input the server vector and the client vector

Here is the input:

In [59]:
```
import random
input_0 = [2,4,6,8,10,1,3]
input_1 = [1, 0, 0, 0, 0, 0, 0]
```

the sever vector

the client vector with the i index the client want to retrieve the ith element of the vector the server holds

Lets quickly do the calculation in the clear so we know what answer to expect:

In [60]:
```
sum = 0
for i in range(len(input_0)):
    sum += input_0[i]*input_1[i]
sum
```

Out[60]: 2

Step 6.2 Start the new job for this retrieval and Use the HE library which is SEAL_BFV

In [80]:
```
'''Step 6.2 Start the new job for this retrieval and Use the HE library which is SEAL_BFV'''

sheep_client.new_job()
```

Out[80]: {'status_code': 200, 'content': ''}

In [84]:
```
''' Use the HE library which is SEAL_BFV'''

sheep_client.set_context("SEAL_BFV")
```

Out[84]: {'status_code': 200, 'content': ''}

Step 6.3 Generate the dot product for this retrieval with the circuit

```
In [66]: circuit = generate_vector_dot_product_circuit(input_0,input_1)
         sheep_client.set_circuit_text(circuit)
```

use the circuit

```
Out[66]: {'status_code': 200, 'content': ''}
```

```
In [67]: sheep_client.set_inputs({"input_0":input_0, "input_1": input_1})
Out[67]: {'status_code': 200, 'content': ''}
```

use the two inputs

```
In [68]: sheep_client.set_const_inputs({"rotate_1": -1})
Out[68]: {'status_code': 200, 'content': ''}
```

set the rotate operation

```
In [69]: sheep_client.set_timeout(60)
Out[69]: {'status_code': 200, 'content': ''}
```

```
In [70]: sheep_client.run_job()
```

run the job

```
Out[70]: {'status_code': 200, 'content': ''}
```

```
In [71]: sheep_client.get_results()['content']["cleartext check"]["is_correct"]
Out[71]: True
```

It means the operations above is successful.

```
In [72]: sheep_client.get_results()['content']["outputs"]["output"][0].split(",")[0]
Out[72]: '2'
```

Eventually, the client get the i th element where i = 0 which is the 1st element in the vector d = [2,4,6,8,10,1,3] the server holds. idex starts from 0 in SHEEP circuit.

So, we get the right answer!

Above is the 1st time: [2, 4, 6, 8, 10, 1, 3] X [1, 0, 0, 0, 0, 0, 0] = **2**+0+0+0+0+0+0 = 2

Repeat this step, the client will get all the elements in the vector d the sever holds.

2nd time: [2, 4, 6, 8, 10, 1, 3] X [0, 1, 0, 0, 0, 0, 0] = 0+**4**+0+0+0+0+0 = 4

```
In [40]: import random
         input_0 = [2,4,6,8,10,1,3]
         input_1 = [0, 1, 0, 0, 0, 0, 0]
```

Lets quickly do the calculation in the clear so we know what answer to expect:

```
In [41]: sum = 0
         for i in range(len(input_0)):
             sum += input_0[i]*input_1[i]
         sum
Out[41]: 4
```

we get the right answer!

3rd time: [2, 4, 6, 8, 10, 1, 3] X [0, 0, 1, 0, 0, 0, 0] = 0+0+**6**+0+0+0+0 = 6

```
In [42]: import random
         input_0 = [2,4,6,8,10,1,3]
         input_1 = [0, 0, 1, 0, 0, 0, 0]
```

Lets quickly do the calculation in the clear so we know what answer to expect:

```
In [43]: sum = 0
         for i in range(len(input_0)):
             sum += input_0[i]*input_1[i]
         sum
Out[43]: 6
```

4th time: [2, 4, 6, 8, 10, 1, 3] X [0, 0, 0, 1, 0, 0, 0] = 2+0+0+**8**+0+0+0 = 8

In [44]:
```
import random
input_0 = [2,4,6,8,10,1,3]
input_1 = [0, 0, 0, 1, 0, 0, 0]
```

Lets quickly do the calculation in the clear so we know what answer to expect:

In [45]:
```
sum = 0
for i in range(len(input_0)):
    sum += input_0[i]*input_1[i]
sum
```

Out[45]: 8

5th time: [2, 4, 6, 8, 10, 1, 3] X [0, 0, 0, 0, 1, 0, 0] = 0+0+0+0+**10**+0+0 = 10

In [46]:
```
import random
input_0 = [2,4,6,8,10,1,3]
input_1 = [0, 0, 0, 0, 1, 0, 0]
```

Lets quickly do the calculation in the clear so we know what answer to expect:

In [47]:
```
sum = 0
for i in range(len(input_0)):
    sum += input_0[i]*input_1[i]
sum
```

Out[47]: 10

6th time: [2, 4, 6, 8, 10, 1, 3] X [0, 0, 0, 0, 0, 1, 0] = 0+0+0+0+0+**1**+0 = 1

In [48]:
```
import random
input_0 = [2,4,6,8,10,1,3]
input_1 = [0, 0, 0, 0, 0, 1, 0]
```

Lets quickly do the calculation in the clear so we know what answer to expect:

In [49]:
```
sum = 0
for i in range(len(input_0)):
    sum += input_0[i]*input_1[i]
sum
```

Out[49]: 1

In [76]:
```
sheep_client.get_results()['content']["outputs"]["output"][0].split(",")[0]
```

Out[76]: '1'

So we get the right answer!

7th time: [2, 4, 6, 8, 10, 1, 3] X [0, 0, 0, 0, 0, 0, 1] = 0+0+0+0+0+0+**3** = 3

```
In [50]: import random
         input_0 = [2,4,6,8,10,1,3]
         input_1 = [0, 0, 0, 0, 0, 0, 1]
```

Lets quickly do the calculation in the clear so we know what answer to expect:

```
In [51]: sum = 0
         for i in range(len(input_0)):
             sum += input_0[i]*input_1[i]
         sum
Out[51]: 3
```

```
In [62]: sheep_client.get_results()['content']["outputs"]["output"][0].split(",")[0]
Out[62]: '3'
```

So we get the right answer!

We can see that the client sends the specific vector s each time to retrieve the ith element in the vector d the server holds successfully.

Here is the second way to retrieve ith element in the vector the server holds via this notebook script.

Step A. This step is the same as Step 4. Use the HE(homomorphic encryption) scheme which is SEAL_BFV and design the circuit with 7 slots

```
In [30]: sheep_client.set_context("SEAL_BFV")
         sheep_client.set_input_type("int8_t")
         sheep_client.get_nslots()
Out[30]: {'status_code': 200, 'content': {'nslots': 2048}}
```

Here is the circuit I designed: circuit-vector-dot-7slots.sheep

Note: Here is the main logic about the dot product that is designed based on ROTATEs and ADDs. With this set of parameters, we have 7 slots available, so can do a dot product of two 7-component vectors. The circuit to perform this operation will be a MULTIPLY followed by a sequence of 6 ROTATEs and ADDs.

```
INPUTS input_0 input_1
CONST_INPUTS rotate_1
OUTPUTS output

input_0 input_1 MULTIPLY prod_r0

prod_r0 rotate_1 ROTATE prod_r1
prod_r0 prod_r1 ADD prod_s1
```

prod_r1 rotate_1 ROTATE prod_r2
prod_s1 prod_r2 ADD prod_s2

prod_r2 rotate_1 ROTATE prod_r3
prod_s2 prod_r3 ADD prod_s3

prod_r3 rotate_1 ROTATE prod_r4
prod_s3 prod_r4 ADD prod_s4

prod_r4 rotate_1 ROTATE prod_r5
prod_s4 prod_r5 ADD prod_s5

prod_r5 rotate_1 ROTATE prod_r6
prod_s5 prod_r6 ADD output

Let's perform the circuit:

In [79]:
```
circuit = """
INPUTS input_0 input_1
CONST_INPUTS rotate_1
OUTPUTS output

input_0 input_1 MULTIPLY prod_r0

prod_r0 rotate_1 ROTATE prod_r1
prod_r0 prod_r1 ADD prod_s1

prod_r1 rotate_1 ROTATE prod_r2
prod_s1 prod_r2 ADD prod_s2

prod_r2 rotate_1 ROTATE prod_r3
prod_s2 prod_r3 ADD prod_s3

prod_r3 rotate_1 ROTATE prod_r4
prod_s3 prod_r4 ADD prod_s4

prod_r4 rotate_1 ROTATE prod_r5
prod_s4 prod_r5 ADD prod_s5

prod_r5 rotate_1 ROTATE prod_r6
prod_s5 prod_r6 ADD output
"""
```

In [80]:
```
sheep_client.set_circuit_text(circuit)
sheep_client.get_inputs()
```

Out[80]: {'status_code': 200, 'content': ['input_0', 'input_1']}

We see the circuit is performed successfully.

Step B. So the two input vectors are called input_0 and input_1. Let's assign them the values {2,4,6,8,10,1,3} and {1,0,0,0,0,0,0}

```
In [80]: sheep_client.set_circuit_text(circuit)
         sheep_client.get_inputs()

Out[80]: {'status_code': 200, 'content': ['input_0', 'input_1']}
```

So the two input vectors are called input_0 and input_1. Let's assign them the values {2,4,6,8,10,1,3} and {1,0,0,0,0,0,0}

```
In [81]: sheep_client.set_inputs({"input_0": [2,4,6,8, 10, 1, 3], "input_1": [1,0,0,0, 0, 0, 0]})

Out[81]: {'status_code': 200, 'content': ''}

In [82]: sheep_client.get_const_inputs()

Out[82]: {'status_code': 200, 'content': ['rotate_1']}
```

The circuit also takes a "const input" (that won't be encrypted) - this is how much we will ROTATE the vector by in each step, so just set it to -1

```
In [83]: sheep_client.set_const_inputs({"rotate_1": -1})

Out[83]: {'status_code': 200, 'content': ''}
```

Step C. Run the job

```
In [84]: sheep_client.run_job()

Out[84]: {'status_code': 200, 'content': ''}
```

Step D. Get the result the client wants retrieve

```
In [10]: sheep_client.get_results()

Out[10]: {'status_code': 200,
          'content': {'cleartext_check': {'is_correct': True},
           'outputs': {'output': ['2,2,2,2,2,2,2']},
           'timings': {'decryption': '755.541000',
            'encryption': '3303.941000',
            'evaluation': '18361.711000',
            'output': '187.871000',
            'prod_r0': '8900.410000',
            'prod_r1': '1391.484000',
            'prod_r2': '1172.884000',
            'prod_r3': '1137.894000',
            'prod_r4': '1090.599000',
            'prod_r5': '1459.108000',
            'prod_r6': '1441.695000',
            'prod_s1': '125.464000',
            'prod_s2': '176.069000',
            'prod_s3': '167.236000',
            'prod_s4': '165.544000',
            'prod_s5': '345.614000'}}}
```

We see the client send [1,0,0,0, 0, 0, 0] and use HE encryption which is SEAL_BFV, the client successfully gets the i element.

Note: the output is ['2,2,2,2,2,2,2'] (we always get an output vector the same length as our input vector, even though in this case we only need one number), and 2 is indeed the scalar product of [2,4,6,8,10,1,3] and [1,0,0,0,0,0,0].

Repeat this step with vector s, the client can retrieve each element of the vector that the server holds.

Way 2 Work on SHEEP Web Interface

1. Develop the circuit for dot-product in the high-level language which is SHEEP language:

The code script is 'hw-4-3-code-circuit-vector-dot-7slots.sheep' in the submission

```
INPUTS input_0 input_1
CONST_INPUTS c_input_0 c_input_1 c_input_2
OUTPUTS output_0
 input_0 input_1 MULTIPLY output_1
 output_1 c_input_0 ROTATE output_2
 output_1 c_input_1 ROTATE output_3
 output_1 c_input_2 ROTATE output_4
 output_1 output_2 ADD output_5
 output_5 output_3 ADD output_6
 output_6 output_4 ADD output_0
```

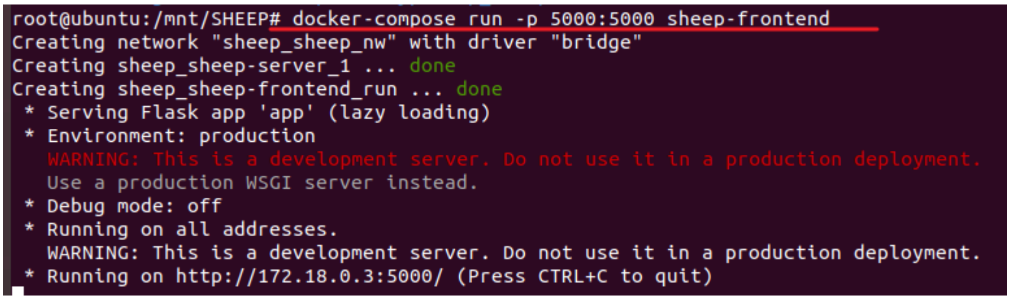
2. Given the client wants to retrieve the sixth element. So, the vector that the client sends to the server should be [0,0,0,0,0,1,0]. Based on the protocol I designed above and the server holds d = [2, 4, 6, 8, 10, 1, 3]. Here is how the client can retrieve the element in this case:
[d_1, d_2, d_3, d_4, d_5, d_6, d_7] X [E(0), E(0), E(0), E(0), E(0), E(1), E(0)] = d_6
Simply to understand, [2, 4, 6, 8, 10, 1, 3] X [0, 0, 0, 0, 0, 1, 0] = 0+0+0+0+0+1+0 = 1
→The sixth element the server holds retrieved by the client is 1.

Note: We can directly apply the HE library provided via SHEEP to encrypt via the web interface. SHEEP project allows the user to execute their own programs via either a web interface or programmatically, the library incorporates a set of native benchmarks.

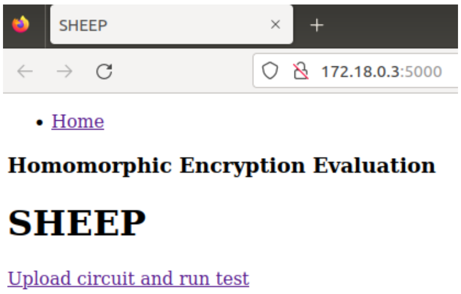
a) If we run the command: docker-compose run -p 5000:5000 sheep-frontend. We can work on the SHEEP web interface (with the circuit)
b) If we run the command: docker-compose run -p 8888:8888 sheep-notebook. We can just work on Jupyter Notebook. Specifically, work on the script 'vector_dot_product.ipynb'

3. If the client repeatedly sends 7 different binary vectors to retrieve each element that the server holds, the client will be able to retrieve all the elements of the vector that the server holds (the number of binary vectors is corresponding to the size of the vector the sever holds).
Simply to understand:
[2, 4, 6, 8, 10, 1, 3] X [1, 0, 0, 0, 0, 0, 0] = **2**+0+0+0+0+0+0 = 2
[2, 4, 6, 8, 10, 1, 3] X [0, 1, 0, 0, 0, 0, 0] = 0+**4**+0+0+0+0+0 = 4
[2, 4, 6, 8, 10, 1, 3] X [0, 0, 1, 0, 0, 0, 0] = 0+0+**6**+0+0+0+0 = 6
[2, 4, 6, 8, 10, 1, 3] X [0, 0, 0, 1, 0, 0, 0] = 2+0+0+**8**+0+0+0 = 8
[2, 4, 6, 8, 10, 1, 3] X [0, 0, 0, 0, 1, 0, 0] = 0+0+0+0+**10**+0+0 = 10
[2, 4, 6, 8, 10, 1, 3] X [0, 0, 0, 0, 0, 1, 0] = 0+0+0+0+0+**1**+0 = 1
[2, 4, 6, 8, 10, 1, 3] X [0, 0, 0, 0, 0, 0, 1] = 0+0+0+0+0+1+**3** = 3
Let's work on [2, 4, 6, 8, 10, 1, 3] X [0, 0, 0, 0, 0, 1, 0] = 0+0+0+0+0+1+0 = 1 on the SHEEP web interface.

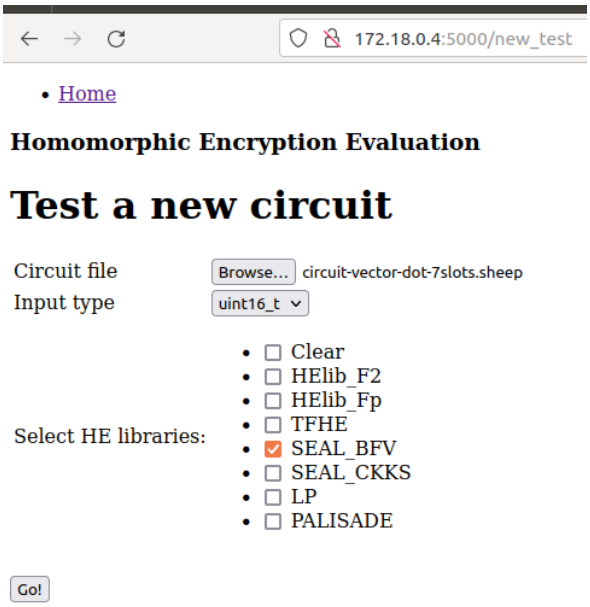
Step 1. Enter the directory the source code installed that has "docker-compose.yml" and run the command: docker-compose run -p 5000:5000 sheep-frontend



Now, the SHEEP server is created and deployed successfully. The frontend (SHEEP web interface) is running.

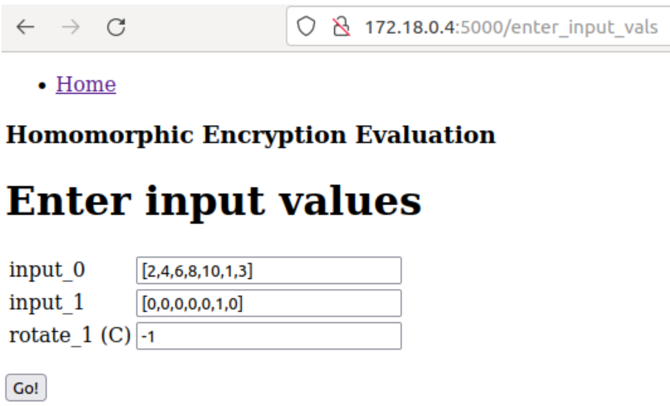


Step 2. Click the link "Upload circuit and run test"

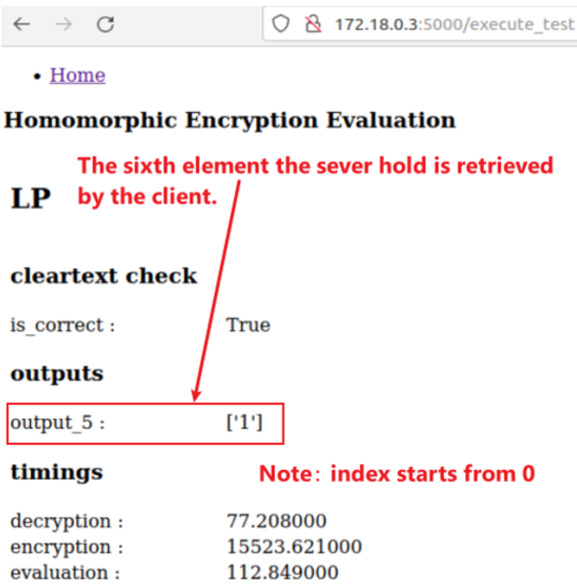


Step 3. Enter Parameters

Step 4. Input the Vectors d and s



Step 5. we get the sixth element that the client wants to retrieve is 1.



Repeatedly, the client can each element based on the protocol I designed above.