1 Using Our Solution

One can use our implementation (an executable program) with ease by providing three files that contain configuration, actual data, and the binary circuit representing the output function f.

1.1 Compilation

In Windows environment, we provide a solution file for Visual Studio 2008/2010. In Linux environment, we provide a Makefile.

1.2 Configuration

Figure 1 describes the configuration for two-party case. The configuration file for the provider (resp., the customer) is inputp.txt (resp., inputc.txt).

The meaning of the fields is straightforward but explained in the following to be complete:

- `num_parties`: It represents the number of parties participating in the protocol
- `pid`: It represents the identifier of the participating party. The id starts from 0, and the customer must have the last id (i.e., num_parties-1).
- `address-book`: It represents the name of the file that contains the addresses of the parties. Each address in the file is of a form (pid, ip-address, port).
- `input`: An input file that contains the party's input values (such as resource values, or 0/1 indicating interests in each resource)
- `num_input`: The number of the party's input values (from the above input file on `input` field) to input to the implementation
- `circuit`: The name of the file that contains the circuit representing a function to be computed.
- `create-circuit`: Instead of specifying a circuit file on `circuit` field, one can also build a circuit with a source-code level with minimal modification on the source code. See Sec. 2.2 and 2.3
- `seed`: It represents a random seed necessary for SHA-1.
- `p`: It is a safe prime number to be used in the oblivious transfer. It must be the same among all the parties. We recommend p to be at least 512 bits long.
- `g`: It is a generator of the subgroup $QR(\mathbb{Z}_p^*)$. It must be the same among all the parties.

In `configp.txt`:

```
num_parties  2
pid 0
address-book addresses.txt
circuit circ.bin
input inputp.txt
num_input 5000
seed 11323434087201
p 893609795...799235731486130087
g 766091584...167705188057761352
```

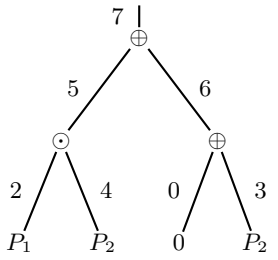
In `configc.txt`:

```
num_parties  2
pid 1
address-book addresses.txt
circuit circ.bin
input inputc.txt
num_input 5000
seed 39347104378941
p 893609795...799235731486130087
g 766091584...167705188057761352
```

In `addresses.txt`:

```
0 192.168.1.4 7766
1 192.168.1.5 7767
```

Figure 1: Configuration Example




```
n 2
d 7 5 2
i 0 2 2
i 1 3 4
o 0 1 0
o 1 7 7
v 0 1
v 1 1
g 0 0 -1 -1 0
g 1 0 -1 -1 0
g 2 0 -1 -1 1 5
g 3 0 -1 -1 1 6
g 4 0 -1 -1 1 5
g 5 1 2 4 1 7
g 6 2 0 3 1 7
g 7 2 5 6 0
```

Figure 2: Circuit Example

1.3 Providing input

The input is provided via the input file given in the configuration (on `input` field). The formation can be any form depending on the circuit that is to be computed (the circuit descriptions are in Sec. 2). The following is an example of a Boolean format:

```
1 0 1 0 1 1 1 ...
```

Each input value is divided by a single-column space, and the number of inputs in the input file should be at least as many as the number given in `num_input` field of the configuration.

1.4 Running the program

Using our implementation is simple. Following the example in the Figure 1, the provider will run:

```
>mpc.exe configp.txt
```

Similarly, the customer will run:

```
>mpc.exe configc.txt
```

2 Providing Circuits

2.1 Circuit as a file

One can use our implementation with ease by providing three files that contain configuration, actual input, and the Boolean circuit representing the function f of interest. A circuit may be described as a file using the fields below. We show an example circuit in Figure 2.

- `n`: It represents the number of parties participating in the protocol.
- `d`: It represents the overall description of the circuit: the total number of wires in the circuit, the id w of the first non-input wire, and the number of XOR gates in the circuit. The wires before w are considered as input wires.

- `i`: It describes the input part of a party in the circuit: party id, the wire id of the first input of the party, and of the last.
- `o`: It describes the output part of a party in the circuit: party id, the wire id of the first output of the party, and of the last. If a party has no output, the last wire id should be smaller than the first. In Figure 2, only the second party P_1 has an output for wire 7.
- `v`: It describes the party id and the number of bits that each item in the input file represents. Since this value is set to 1 in Figure 2, the MPC engine will read data from the input file and treat each tokenized value as a 1-bit integer when feeding this input values to the circuit.
- `g`: It represents the gate. The id of each gate is the id of the output wire of the gate. The values to be specified are the gate id, the gate type, the left-input wire id, the right-input wire id, the number of parents (i.e., gates that take the output of this gate as input). The gate type can be either input (0), AND (1), XOR (2). If the number of parents of the given gate is non-zero, the ids of the parents are listed. Note that the first two gates are reserved for constant 0 and 1; the circuit file should follow this rule.

Converting the text file into a binary one. To improve performance by avoiding parsing overhead, our MPC implementation recognizes only the binary file. Conversion can be done as follows:

```
>mpc.exe -c circ.txt circ.bin
```

2.2 Default circuits

The following default circuits are embedded in the source code. To test these circuits, one should specify `create-circuit` field in the configuration. There are currently three circuits we provide whose names are `p2p`, `cloud`, and `social-net`, respectively.

Circuit description for `p2p`. The circuit considers the case where a single customer P_{n+1} interacts with a set of providers $\mathbf{P} = \{P_1, P_2, \ldots, P_n\}$. Each provider P_i holds (private) input x_i, and the customer holds (private) input x_{n+1}. The customer will compute $f(x_1, \ldots, x_n, x_{n+1})$. The input of each provider P_i is a vector of resource values $x_i = \{v_r\}_{r \in R_i}$, where $R_i = [(i-1)k + 1, ik]$ (k is specified as `num_input` in the configuration file). The input of the customer is the vector $x_{n+1} = (b_1, \ldots, b_{nk})$ where $b_i \in \{0, 1\}$ for $i \in [1, nk]$. Let $i_r = v_r b_r$. The output of the circuit is a resource r maximizing i_r.

To test `p2p` circuit, one should specify `create-circuit` field in the configuration of every party. One example is as follows:

```
create-circuit p2p 2500 65535
```

The first argument indicates the name of the circuit to test, and the second is the number of resources each provider inputs, corresponding to k. The third argument indicates that the range for each input value is 0 $\sim$ 65535. This third argument is required to decide the number of bits to represent each input value. In this example, every input from the parties is represented by 16 bits.

The definition for each argument depends on the circuits, and therefore other circuits such as `cloud` and `social-net` can have different arguments (see Sec. 2.3).

Saving the default circuit into files. The default circuit can be save as a text file:

```
>mpc.exe config.txt -t circ.txt
```

Also, one can get a binary circuit file.

```
>mpc.exe config.txt -b circ.bin
```

2.3 Modifying the source code

If the size of a circuit becomes more than 10^6, loading the circuit from a file takes much longer than creating it directly within the program. To achieve higher performance, one can also customize the circuit with a source-code level with minimal modification; the MPC engine in our implementation refers to circuits via an abstract C++ interface. To build a circuit on the source code, follow the next steps:

1. Make a new circuit class, for example `CNewCircuit`, that inherits `CCircuit` class.
2. Override the following `Create` interface defined in `CCircuit`,

   ```
   virtual BOOL Create(int nParties, const vector<int>& params) = 0;
   ```

   Note that on `create-circuit` field all the arguments except the first one (which is the circuit name) correspond to vector `params`.

   - Build the new circuit so that all the gates of the circuit can be saved in `m_pGates` (`Gate` structure array defined in `CCircuit`).
   - Gates for inputs: record the id's of the starting and the ending gates among all input gates that take all inputs of each party in vector `m_vInputStart` and `m_vInputEnd` (defined in `CCircuit`) respectively. All input gates should come right after the two gates reserved for constant bits 0 and 1. In other words, the input gates start with id 2.
   - Gates for outputs: record the id's of the starting and the ending gates among all output gates that return all outputs for each party in vector `m_vOutputStart` and `m_vOutputEnd` (defined in `CCircuit`) respectively. All output gates should come right after the rest of the entire circuit gates. If a party does not require any output, `m_vOutputEnd`'s element for that party should be smaller than `m_vOutputStart`'s element for that party. Then no output for that party will be returned.
3. Add `CNewCircuit` to `CCircuit` by simply defining this new class at `CREATE_CIRCUIT` function in `CCircuit`.