SPDZ Tutorial: Getting Started

Chan Nam Ngo
channam.ngo@unitn.it
University of Trento, Trento, Italy

December 19, 2018

1 Intallation of the SPDZ library

All the tutorials assume the Linux platform (e.g. Ubuntu). Students using other platforms should find equivalent alternatives. An easy solution would be VirtualBox[1] + Ubuntu[2].

One should follow the instruction on `https://github.com/bristolcrypto/SPDZ-2`[3] for installation.

2 Introduction to the SPDZ library

Details of the library can be found on the main repository page at `https://github.com/bristolcrypto/SPDZ-2`.

Basically SPDZ is a library that provides a programming framework for Secure Multiparty Computation (MPC). Moreover, it includes a virtual machine that executes programs in a specific bytecode. One can program the desired computation using high-level Python code which will be compiled and optimized with a particular focus on minimizing the number of communication rounds (for protocol based on secret sharing) or on AES-NI pipelining (for garbled circuits).

As an example, the program below is used to compute the millionares problem, i.e. Alice and Bob each has a secret cash amount and they want to find out who is richer without the other knowing their secret amount.

```
program.bit_length = 32

def millionnaires():
        """ Secure comparison, receiving input from each party via stdin """
        print_ln("Waiting for Alice's input")
        alice = sint.get_input_from(0)
        print_ln("Waiting for Bob's input")
```

[1] `https://www.virtualbox.org/wiki/Downloads`
[2] `https://www.ubuntu.com/`
[3] SPDZ-2 is now inactive but is still useful for a demonstration of MPC Software.

```
bob = sint.get_input_from(1)

b = alice < bob
print_ln('The richest is: %s', b.reveal())
```

3 Notes on SPDZ

3.1 Data Type

All data types can be found at `https://github.com/bristolcrypto/SPDZ-2/blob/master/Compiler/types.py`. As a note the prefix *"c"* refers to *clear* data types, e.g. *cint*, meaning the value is public while the prefix "s" refers to *secret*, e.g. *sint*, which means the value is secret.

A secret value can be *revealed* meaning the value is available to all parties. As an example in the millionares program above, *b = alice ¡ bob*, which is the output of the program, can be opened to Alice and Bob for them to find out the final result by calling *b.reveal()*.

Data types also include array type, e.g. *Array(N, sint)* is an *sint* array of size N.

3.2 Getting secret inputs

Secret inputs can be obtained via stdin, e.g. *alice = sint.get_input_from(0)*, or from a file, e.g. *alice = sint.get_raw_input_from(0)*.

As an example the two millionares problem above can be extended to N millionares.

```
from util import if_else

def millionnaires(N):
        V = Array(N, sint)
        R = Array(N, sint)
        for i in range(N):
                inps = [sint.get_raw_input_from(i) for _ in range(2)]
                V[i] = inps[0]
                R[i] = sint(0)

        m = V[0]
        for i in range(N):
                m = if_else(V[i] >= m, V[i], m)
        for i in range(N):
                R[i] = if_else(V[i] >= m, sint(1), sint(0))

        for i in range(N):
                print_ln('Output for %s is: %s', i, R[i].reveal())
```

millionnaires(256)

3.3 Statements

In the example above, the *for* loop is executed in clear while *if_else* is a secure computation. The supported statements can be found at `https://github.com/bristolcrypto/SPDZ-2/blob/master/Compiler/library.py`[4].

As an example, the following is extracted for if-then-else statement.

```
def if_then(condition):

def else_then():
try:
if state.has_else:
# run the else block

def end_if():
# start next block
if state.has_else:
# jump to else block if condition == 0
# set if block to skip else
else:
# set start block's conditional jump to next block
# nothing to compute without else

def if_statement(condition, if_fn, else_fn=None):
if condition is True or condition is False:
# condition known at compile time

def if_(condition):

def if_e(condition):

def else_(body):
```

4 Getting Started

Implement the following problems using SPDZ.

- The *Secure Difference Detection* runs on common input f and interacts with a set of players $(P_1, \ldots, P_N)$ and receive (f_i, r_i) from each P_i. Upon receiving all inputs, let c_f be the number of pairs $(f_i = f)$, if $c_f > 0$ output $y = \sum r_i \bmod c_f$ to all players and $\perp$ otherwise.

[4]The defined functions and actual code usage may look different due to Python syntax.

- The *Secure Threshold Comparison* runs on common input (t) and interacts with a set of players $(P_1, \ldots, P_N)$ and receive (η_i, r_i) from each P_i. Upon receiving all inputs, let c_f be the number of pairs $(\eta_i < t)$, if $c_f > 0$ output $y = \sum r_i \bmod c_f$ to all players and $\perp$ otherwise.