Using Yao's Protocol to Calculate Common Elements of two Sets

Friedrich Hartmann

June 15, 2024

Contents

1 Overview

In order to calculate the common elements for two parties, Alice and Bob, I used garbled circuits such that neither Alice nor Bob reveal elements they don't share. Each set is read from a text-file containing comma separated numbers that are interpreted as 32-bit floats.
The implementation is a wrapper around a Secure Multi-Party Computation library which can be found on GitHub.

1.1 Protocol Description

The protocol works as follows.

3. Alice and Bob both shuffle their inputs.
4. Alice and Bob communicate their number of elements, k and l, in each set.
5. Yao's Protocol is executed $\mathcal{O}(k \cdot l)$ times to pairwise test the equality of two elements. I the latter case, both parties add that element to the set of shared elements.

At first, I had to decide whether to create a boolean-circuit that calculates a set intersection in one go or to perform an iterative pairwise test to achieve the same. The first has the advantage that there is less communication between Alice and Bob for a price of a rather complicated circle. Furthermore, the exact positions, where the elements are equal, are not revealed.

However, I decided to take the second approach, as I wanted to automate the circuit generation process in order to theoretically support the encoding of arbitrary length integers or floating point numbers. Furthermore, I want to hide any information about the ordering of a set. That's why the inputs are permuted at first.

1.2 Circuit Definition

At first, I implemented a circuit-generation function that outputs a circuit in json format which checks whether two n-bit sequences a, b are equal. The circuit has $2n$ input wires. The bits a are mapped to the wires $\{1, ..., n\}$ while b is mapped to the wires $\{n+1, ..., 2n\}$. There is only one output wire that is 1 in case of $a = b$ and otherwise 0.

The length n is restricted to be a power of 2, because the circuit has a binary-tree like structure. Input wires are mapped to leaves in such a way that a pair of leaves (a_i, b_i), with $i \in \{1, .., n\}$, $a_i \in a$, $b_i \in b$, has a $NXOR$-gate as a parent. The bits a_i and b_i share the same position i in a and b, respectively. Such a gate outputs 1, in case both bits are equal and otherwise 0. All the other nodes are AND-gates that encapsulate the predicate that a_i and a_b have to be equal for all positions i if $a = b$.

1.3 Design

The general idea is to have two scripts *alice.py* and *bob.py* that call the original library as a sub-process from different shells. Those scripts manage to read the sets from file and to execute the protocol as described in the previous section. The result of each call is obtained by reading *stdout*.

1.4 Functionalities

1.4.1 Input Parameter

I added a command line argument $-i$ to the original garbled-circuit code base which allows to specify the assignment of the circuit's input wires for Alice and Bob. Before that, all possible combinations of inputs were calculated and printed as a table. I adjusted it to only consider the case that is determined by the given inputs.

1.4.2 Logger

As stated before, the process of forwarding the results from evaluating the garbled circuit is done via *stout*. Hence, the terminal cannot be used to log any communication from Yao's protocol. That's why I write the log statements to files that can be found in *set_intersection/logs/*.

1.4.3 Utilities

The file *util.py* contains helper functions that

1. read the respective sets from a file.
2. perform the conversion between floating point numbers and bit-strings of length 32.
3. abstract from a channel on which the set length is communicated. I did not implement that channel but give interfaces in order to receive the length.

Along with that, I specify constants which define the supported bit length and the location of the files containing the sets.

2 How to Run

See *README.md* in the project's root directory.

3 Limitations

The protocol, as it is, cannot be used to perform the calculation on the network. That's because I did not implement a channel on which the length of each set is exchanged. But, to know the number of calls to Yao's protocol, this is necessary . Therefore, the execution works only locally.

Second, the gate is fixed to handle input of length 32 which is interpreted as float for each party. Since I can generate arbitrary big gates for input of length 2^n, the support for any type of number can be implemented easily.

4 Optimizations

The runtime can be optimized by changing the behavior of Alice and Bob in the original code base. Bob runs as a server and Alice connects as a client. Both are executed with a fixed assignment to the wires. In consequence, both the server and the client are shut down after evaluating one equality between two elements and are run with new inputs again and again. So, letting the server and client be connected while updating the inputs results in a speed up.