Multi-Party Computation Assignment

October 24, 2019

1 Introduction

Multi-party computation is the generalization of two-party computation. Instead of garbled circuits, we use a technique based on secret sharing. The idea is that we store a confidential dataset of secret shared values, and can run queries and computations on it without revealing the data.

In this programming assignment, you'll implement routines for encoding and decoding secret shared data, a multi-party computation protocol, and finally several MPC programs.

This document gives a short but mostly self-contained overview of the protocol and outlines the tasks to complete in the assignment, to serve as a checklist. The comments in the provided skeleton code also follow this closely.

2 Multi-Party Computation Protocol

Secret sharing MPC is about computing on secret shared data. A secret sharing of $x \in \mathbb{F}_p$ is a way of encoding x in such a way that N nodes each store a different share (node i stores $[\![x]\!]^{(i)}$. The idea is that confidentiality, integrity, and availability guarantees should hold even if up to f of the nodes are compromised.

We use the Shamir secret sharing scheme. To encode a piece of secret data $x \in \mathbb{F}_p$, where $\mathbb{F}_p$ is a finite field of prime order-p, we first generate a random degree-f polynomial ϕ, such that $x = \phi(0)$. Each node i stores their share of the data as $[\![x]\!]^{(i)} = \phi(i)$. The data can be recovered from a subset of $f + 1$ shares from honest parties through Lagrange interpolation (though more thought is needed to handle Byzantine errors).

Multiparty computations We can write a variety of programs to operate on secret shared data. For example, given secret shared inputs, $[\![x]\!], ...$, we may want to evaluate a program prog to get $[\![o]\!] = [\![\mathsf{prog}(x, ...)]\!] = \mathsf{execMPC}(\mathsf{prog}, [\![x]\!], ...)$ and then run $o \leftarrow \mathsf{open}([\![o]\!])$ to publish just the output of the program.

MPC programs like prog are written in a uniform way, even though they are executed in a replicated fashion among the N different nodes. Linear operations on secret shared data can simpliy be performed locally at each node. The opening operation, $x \leftarrow \mathsf{open}([\![x]\!])$ requires communication between all the nodes. In this assignment, the skeleton code includes a simple MPC simulator, which runs N instances of a program prog in separate (green-)threads, and routes messages between them when encounter an open instruction. This simulator makes use of the secret sharing encoder and decoder you complete, but otherwise should work already.

To build useful programs on top of MPC, we often need to make use of pre-processing data. The challenge is that although we can perform linear operations on secret shared data trivially, we need to use a variety of clever algorithms to do non-linear operations. For example, given a secret value $[\![x]\!]$ and a random preprocessing value $[\![r]\!]$, publishing $\mathsf{open}([\![rx]\!])$ reveals no information about x, but can be used to compute $[\![x^{-1}]\!]$, which would otherwise be difficult. The MPC simulator provides easy access to a stash of random preprocessing values.

Below we describe the required tasks in detail, in the order we recommend approaching them. You can also search for the string `TODO` in the python files for hints on where to begin.

Skeleton Code Explanation The skeleton code consists of the four files `polynomials.py`, `secretsharing.py`, `mpc_sim.py`, and `fouriertransform.py`. Each of these has clearly marked Problems and TODO: hints to suggest where to include your code.

How to run the code You can run each file directly to see some test cases run, e.g. `python secretsharing.py`.

3 Tasks

1. **Interpolating Polynomials** [15pts] We will build on the same library for polynomials over finite fields that we used in MP1. This provides operations like coefficients, including adding and multiplying polynomials

together. However we need to implement other operations. The file `polynomials.py` contains tasks to complete these additional functions. As a warmup, the first task you'll need to complete is polynomial evaluation: given the coefficients for $\phi(\cdot)$, and a value for x, evaluate $\phi(x)$. Use Horner's rule to do it efficiently.

The next task is about Lagrange polynomials. Given a set of values $x_0, ...x_k$, compute the lagrange polynomials $\ell_i(x)$ for $0 \leq i \leq k$.

The lagrange polynomials can be used to perform polynomial interpolation. Given k+1 points $(x_0, y_0), ..., (x_k, y_k)$, find a polynomial ϕ of (at most) degree-k that coincides with each of these points, $y_k = \phi(x_k)$.

The `secretsharing.py` file defines a particular field $\mathbb{F}_p$ we'll use for this assignment — it's chosen for a few special properties we'll get to later. Given polynomial interpolation, we can immediately reconstruct a shared value from all n shares. This is provided for you, you can go straight to `mpc_sim.py` now, but we'll come back to `secretsharing.py` in Problem 3.

2. **MPC Applications** [30pts] The `mpc_sim.py` file contains a simulator for MPC protocols, which is mostly complete (once you finish the interpolation routines, it will mostly work).

   Your task will be implement MPC programs that compute on secret shared values. Secret shares are ordinary field elements, so linear operations on the field elements correspond to linear operations on the shared values. Secret shared values can be opened, making use of the reconstruction protocols above. The MPC applications are presented as puzzles, although we've gone over the main ideas in lectures.

   (a) Beaver Multiplication. [5pts] Just multiplying the shares together, $[\![x]\!] \cdot [\![y]\!]$, would give a degree-$2t$ polynomial, which can't be recovered directly. Beaver Multiplication approch makes use of a preprocessed "beaver triple" of shared values, $[\![a]\!], [\![b]\!], [\![ab]\!]$, which are used once in this protocol and then discarded.

   (b) Computing the inverse of a shared value. [5pts] We start with a secret shared value $[\![x]\!]$ as input, and want to obtain a share of $[\![1/x]\!]$. This is a nonlinear operation. Not only does $1/[\![x]\!]$ not work, we can't in general invert the polynomial ϕ at all (polynomials form a ring, not a field). We can make use of a pre-processing random share $[\![r]\!]$ where r is a random field element.

   (c) Generating a random bit. [5pts] Although we are assuming we have a supply of random field elements $[\![r]\!]$ where r is uniform in the field $\mathbb{F}_p$, we often want to generate a random bit specifically, $[\![b]\!]$ where $b \xleftarrow{\$} \{0, 1\}$. Try to think of a way to achieve convert $[\![r]\!]$ into $[\![b]\!]$ without revealing b.

   (d) Computing powers in constant rounds. [10pts] Given $[\![x]\!]$, we'd like to compute $[\![x^2]\!], ...[\![x^\ell]\!]$, the first ℓ powers of $[\![x]\!]$. Clearly this can be done sequentially using beaver multiplication. But this requires many round trips — round trip latency can be the bottlenck in a distributed system. Can we trade throughput for latency, and compute these in constant round?

   (e) Extracting random values. [5pts] Suppose we have shared values, $[\![x_1]\!], ..., [\![x_n]\!]$, each contributed by a different party. Honest parties generate these randomly and forget about them, but up to f of them may be adversarially chosen, and we don't know which are which. Can we combine these "sketchy" shared values, so that we have $N - f$ "good" shared values, which are guaranteed to be random and unknown?

3. **Robust Reconstruction of Secret Shares**[10pts]

   However, we'll need a more robust approach in the case that some nodes crash or are compromised. The main insight is that if $2f + 1$ shares are recovered, at least $f + 1$ of them must be honest by assumption, and a degree-f polynomial is fully determined by $f + 1$ points. Hence if we can find a degree-f polynomial that coincides with $2f + 1$ points, we'll know it's the correct one.

   Your task is to complete a function that searches for a suitable polynomial by brute force, trying subsets of $f + 1$ vaues at a time and checking if they work.

   Note: More efficient approaches are possible here as well, for example the Berlekamp-Welch method for decoding. This is left as optional suggestion for bonus points, brute force will work for our test cases.

   You can complete the MPC applications just with the simple reconstruction, so the recommended order is to leave robust reconstuction for later.

4. **Discrete Fourier Transform on Finite Fields** [20pts] The fourier transform isn't just for real numbers. We can use the fast fourier transform over finite fields to accelerate computations involving polynomials.

   Suppose we have an element in the field ω that is a *primitive* 2^k*-th root of unity.* This means that $|\omega| = 2^k$, so k is the smallest number such that $\omega^{2^k} = 1$. Then given coefficients of a polynomial $\phi(x) = a_0 + a_1 x + ... + a^{2^k - 1}$, the Discrete Fourier Transform (DFT) is defined as the evaluation of the polynomial at the points $\phi(\omega^0), \phi(\omega^1), ..., \phi(\omega^{2^k - 1})$.

The field $\mathbb{F}_p$ is chosen for a particular property that makes it amenable to do secret sharing using DFT. The order of the multiplicative group, ${\mathbb{F}_p}^\times$, $p-1$, is divisible by a large power of 2, $2^{16}|p-1$. This means that we are able to find 2^n-roots of unity. Your first task is to give an algorithm to find one.

Given a field and an nth root of unity, we can Fast Fourier Transform (FFT) algorithm to compute DFT, and hence evaluation of a degree-n polynomial at n distinct points, in only $O(n \log n)$ time. This can be a big computation savings compared to $O(n^2)$ which is what the earlier solutions take.

Reference: "The (finite field) Fast Fourier Transform"[1]

5. Additional directions. See `mpc_sim.py` for additional challenges to earn bonus points.

[1] https://pdfs.semanticscholar.org/7a2a/4e7f8c21342a989d253704cedfb936bee7d7.pdf