Applications of MPC

Notes by Henry Corrigan-Gibbs

MIT - 6.5610
Lecture 18 (April 10, 2024)

> ***Warning:*** This document is a rough draft, so it may contain bugs. Please feel free to email me with corrections.

Outline

- Reminder: Secure multiparty computation
- Client-server model
- Private aggregation
- Security against malicious clients

Introduction

As a motivating application, consider the following problem: a web-browser vendor wants to know *how many* of its users use incognito mode (or private-browsing mode) without learning which users use the feature.

We know in principle that we can solve this problem with secure multiparty computation: every user holds a bit (whether they used incognito mode or not) and we want to count up the number of "1" bits.

This is an example of a "private-aggregation" problem. Private aggregation shows up often when we want to collect telemetry/usage data in a privacy-protecting way.

Concrete efficiency of multiparty computation

So far, we have looked at secure multiparty computation from a theoretical perspective. In this view:

1. we model a computation as an arithmetic circuit (i.e., a circuit with $+$ and $\times$ gates modulo p) and
2. we treat all parties as having point-to-point secret and authenticated communication channels.

In practice, each of these two simplifying assumptions is problematic:

1. Many computations do not naturally have "nice" representations as arithmetic circuits. In particular, a Python program that runs in time T could blow up into a circuit of size roughly T^2 or worse. So if your Python program runs in 2^{20} steps, which would take roughly one millisecond on your laptop to execute, you could end up with a circuit of roughly 2^{40} gates. To execute the BGW protocol to securely compute this function among n parties would then require each party to transmit roughly n terabytes of data.

   Here we are not even accounting for the fact that executing *one* step of a Python program might take millions of gates to simulate in a circuit. So the true costs of this multiparty computation could be much worse.

2. If the number of parties is in the millions, it may be infeasible for them all to have point-to-point communication links with each other. It would in principle be possible to proxy all of the party-to-party communication through a central server, but in a dynamic environment such as the Internet, it may be difficult to even nail down who the parties *are*, much less have them all agree on shared secrets with each other.

So, if we want to use secure multiparty computation in practice, what are we to do? We are going to cheat in two different ways.

Avoid large circuits: Handle only linear functions

As we discussed when we talked about BGW, computing *linear functions* is easy in a multiparty setting: computing such functions does not even require any communication between the parties, apart from that required to secret-share the data at the start and reconstruct it at the end.

A linear function is just a circuit that only has addition gates and multiplication-by-constant gates.

If we know how to do secure multiparty computation for linear functions very efficiently, let's just do that and only that.

It turns out, with a few tricks caveats that we will not get into, that computing linear functions is already enough to compute a bunch of useful statistical functions:

- SUM, AVERAGE
- VARIANCE,
- MOST POPULAR (approximate),
- ...

For the rest of this lecture, we will focus on variants of the simplest case: every client $i \in \{1, \ldots, n\}$ has a secret value $x_i \in \mathbb{Z}_p$ and we want to securely compute the sum $\sum_{i=1}^{n} x_i \in \mathbb{Z}_p$, where p is a prime larger than the number of parties n.

As one example of this functionality: Computer scientists at BU used a multiparty computation for this function to compute aggregate salary data for Boston-area companies.

Avoid client-to-client interaction: The client/sever model

The next performance cheat that we will use gets rid of the need for client-to-client communication entirely. This is convenient because the set of clients may evolve quickly over time (before even a single iteration of the multiparty protocol has completed) and also because communication is costly.

The idea is to shift most of the work of running the multiparty computation onto a small number of powerful parties ("servers")—maybe only two or three. Each client secret-shares their input to the $k \ll n$ servers, who run the multiparty computation amongst themselves and return the answer to the clients.

This approach has major benefits in terms of communication cost: Each client only communicates with the k servers and each client's communication cost is *independent* of the number of clients.

Compare this per-client cost with that of BGW.

In addition, we only need to worry about keeping the $k \ll n$ servers online in steady state. To participate in the computation, each client only needs to stay online long enough to submit shares of its input to the servers.

We might have a few large tech companies run the servers here.

The downside of this approach is security: An attacker now only needs to compromise $k \ll n$ servers to violate the privacy of (i.e., recover the secret inputs of) all clients. For example, if $k = 2$, then if the two servers decide to get together and share their information, they learn the private data of all clients.

We will ask for two security properties:

Privacy against malicious servers and clients. Intuitively, we want that a coalition of up to $t = k - 1$ servers and any number of clients cannot learn anything about the honest clients' inputs except their sum.

More formally, for every p.p.t. adversary $\mathcal{A}$ controlling at most $t = k - 1$ servers and any number of clients, there exists a p.p.t. simulator $\mathcal{S}$ that can simulate the adversary's view in the real protocol given $f(x_1, \ldots, x_n)$.

This is just a specialization of the general MPC definition to our setting.

Security against malicious clients. Intuitively, we want to make sure that any number of colluding malicious clients cannot mess up the computation of the aggregate statistic. Another way to say this is that the "only thing" that malicious clients can do to affect the protocol output is to choose their own inputs.

More formally, for every p.p.t. adversary $\mathcal{A}$ controlling any number of clients in the real world, there exists a p.p.t. simulator $\mathcal{S}$ controlling the same subset of clients in the ideal world such that for all inputs $x_1, \ldots, x_n$,

$$\mathsf{Real}_{\mathcal{A}}(x_1, \ldots, x_n) \overset{c}{\approx} \mathsf{Ideal}_{\mathcal{S}}(x_1, \ldots, x_n),$$

where $\mathsf{Real}_{\mathcal{A}}(x_1, \ldots, x_n)$ is the output of the *honest parties* after running the real protocol and $\mathsf{Ideal}_{\mathcal{S}}(x_1, \ldots, x_n)$ is the output of the honest parties (with the adversarial parties controlled by the simulator) after all parties hand their inputs to an oracle that returns $y = f(x_1, \ldots, x_n)$ to all parties.

A semi-broken scheme

Back to our private-aggregation example: each client i holds a bit $x_i \in \{0, 1\}$ and the servers want to compute the sum of these bits $\sum_{i=1}^{n} x_i$ over the integers.

To run a multiparty computation for the linear function

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i \quad \in \mathbb{Z}_p$$

among n clients and k servers in the client/server model:

- **Clients: Secret-share inputs.** Each user $i \in \{1, \ldots, n\}$ secret shares their input x_i into k shares and sends one share to each server. If we are using k-out-of-k secret sharing, we can use simple additive secret-sharing instead of Shamir.

  That is, each client i chooses $x_{i1}, \ldots, x_{ik} \quad \in \mathbb{Z}_p$ to be independent uniform random values in $\mathbb{Z}_p$ subject to the constraint

  $$x_i = x_{i1} + \cdots + x_{ik} \in \mathbb{Z}_p.$$

  To each server $j \in \{1, \ldots, k\}$, the client sends the secret share $x_{ij} \in \mathbb{Z}_p$.

  Everything we discuss here will also work with Shamir secret-sharing or any other "linear" secret-sharing scheme.

- **Servers: Run multiparty computation.** The k servers run a k-party computation to compute the value $y = f(x_1, \ldots, x_n) \in \mathbb{Z}_p$.

  In particular, each server $j \in \{1, \ldots, k\}$ just broadcasts the sum of the shares it received:

  $$y_j = \sum_{i=1}^{n} x_{ij} \quad \in \mathbb{Z}_p.$$

  The servers reconstruct the output as $y = y_1 + \cdots + y_j \in \mathbb{Z}_p$.

- **Servers: Broadcast output.** (If necessary.) The servers send the output y back to the clients.

Concrete efficiency. Each client transmits only k elements in $\mathbb{Z}_p$, independent of the total number of clients. Each server transmits a single $\mathbb{Z}_p$ element. This is really cheap.

Privacy against malicious servers. To show security, we need to show that there is a simulator $\mathcal{S}$ that can simulate the adversary's view of the real protocol execution. The simulator here just picks uniform random values for each of the messages it receives from honest parties subject to the constraint that the sum of all of these values is equal to $f(x_1, \ldots, x_h)$, where $(x_1, \ldots, x_h)$ are the honest parties' inputs.

Security against malicious clients. This scheme actually does NOT provide security against malicious clients, in the way we defined. Each client is supposed to submit a value in $\{0, 1\}$ but a malicious client can submit any value in $\mathbb{Z}_p$, where p is large.

In particular, consider an adversarial client that sends a secret sharing of a value $r \in \mathbb{Z}_p \setminus \{0, 1\}$. If we let client n be the adversarial one, the output of the protocol will now be $r + \sum_{i=1}^{n-1} x_i$, for some potentially large value r.

There is no way to simulate this malicious client's behavior in the ideal world: there is no input in $\{0, 1\}$ for the adversarial party that can "explain" the protocol's output in the real world.

Protecting against malicious clients with zero-knowledge proofs on secret-shared data

The problem that our simple multiparty protocol has is that any client can submit shares of a value $x \notin \{0, 1\}$ and the servers have no way to know that this has happened.

We will use a new type of zero-knowledge proof to protect against this sort of misbehavior. The basic idea is that after the client sends shares of its private input to the servers, the client will *prove* to the servers in zero knowledge that the servers' shares add up to a value in $\{0, 1\}$. Only after the servers verify this proof will they accept the client's input. In this way, the servers can protect against any fishy behavior by the client.

Large-scale deployments of private aggregation, by Apple, Google, Mozilla, and others, use these zero-knowledge proofs on secret-shared data to protect against malicious clients.

The setup.

We have k servers/verifiers $V_1, \ldots, V_k$, each holding one value in $z_1, \ldots, z_k \in \mathbb{Z}_p$, and we have a client/prover P holding the value $z = \sum_{i=1}^{k} z_i \in \mathbb{Z}_p$ The verifiers either communicate over point-to-point channels, as in BGW, or potentially a broadcast channel. We also have

a language $\mathcal{L} \subseteq \mathbb{Z}_p$ of "valid" inputs. In our private-aggregation setting, we take $\mathcal{L} = \{0, 1\} \subseteq \mathbb{Z}_p$.

The client's goal is to prove to the servers that its input is valid—that is, that $z \in \mathcal{L}$.

The security goals.

Let $\langle P, V \rangle(\bar{z})$ denote the output of verifiers $V_1, \ldots, V_k$ on inputs $\bar{z} = (z_1, \ldots, z_k)$ at the end of the protocol execution.

As in the zero-knowledge proofs we have seen, we want three properties to hold. The following properties are *information theoretic*: they require no computational assumptions.

- **Completeness.** If all parties are honest, the verifiers accept.
  That is, for all $\bar{z}(z_1, \ldots, z_k) \in \mathbb{Z}_p^k$ such that $z = \sum_{i=1}^k z_i \in \mathcal{L}$,
  $$\langle P, V \rangle(\bar{z}) = 1.$$

- **Soundness.** If the verifiers are honest, then the verifiers reject a cheating prover's input.
  That is, for all $\bar{z} = (z_1, \ldots, z_k) \in \mathbb{Z}_p^k$ such that $z = \sum_{i=1}^k z_i \notin \mathcal{L}$, and all malicious provers P^*,
  $$\Pr[\langle P^*, V \rangle(\bar{z})] \leq \text{“small”}.$$

- **Zero knowledge.** A malicious strict subset of verifiers learns nothing about the honest client's input, apart from the fact that it is in $\mathcal{L}$.
  That is, for all adversarial coalitions of verifiers $V^* \subset \{V_1, \ldots, V_k\}$, where $|V^*| < k$, there exists a simulator $\mathcal{S}$ such that for all $\bar{z} = (z_1, \ldots, z_k) \in \mathbb{Z}_p^k$ such that $z = \sum_{i=1}^k z_i \in \mathcal{L}$,
  $$\mathcal{S} \overset{s}{\approx} \{\mathsf{View}_{V^*}(\bar{z})\},$$
  where $\mathsf{View}_{V^*}(z_1, \ldots, z_k)$ is the adversarial verifiers' view of the protocol execution with P and the honest verifiers on inputs $\bar{z}$.

Here, $\overset{s}{\approx}$ denotes statistical closeness of two probability distributions.

The proof system.

The basic idea is a flavor of a technique due to Ishai, Kushilevitz, Ostrovsky, and Sahai called "MPC in the head." We will only sketch the proof system informally here.

The prover and verifiers will cook up an arithmetic circuit that outputs "o" if and only if its input is "valid." That is, the proof system will be relative to a circuit $C \colon \mathbb{Z}_p \to \mathbb{Z}_p$ such that for all $z \in \mathbb{Z}_p$, we have that $C(z) = 0 \Leftrightarrow z \in \mathcal{L}$. As we have already seen, if $\mathcal{L}$ is

poly-time computable, then we can cook up a poly-size circuit C that has this property.

When $\mathcal{L} = \{0, 1\}$, we have $C(z) = z(z-1) \in \mathbb{Z}_p$.

Since we have already seen BGW, let's tweak the definition slightly to accommodate BGW. We will require security to hold only against $t < k/2$ malicious verifiers and we will replace additive secret sharing everywhere with Shamir secret sharing. The basic idea is otherwise the same.

At the start of the protocol, the verifiers hold t-of-k secret shares of the prover's input $z \in \mathbb{Z}_p$.

- The prover *simulates* the BGW multiparty computation between the k verifiers, computing the value $C(z)$ on their inputs..

  At the conclusion of this simulation, the prover sends to verifier V_i, for $i \in \{1, \ldots, k\}$, a transcript of its view in the simulated BGW protocol execution.

- Each verifier V_i, for $i \in \{1, \ldots, k\}$ first checks that its transcript is *locally consistent*: the verifier checks that its input z_i is equal to its input in the simulated execution. Then the verifier runs through the transcript, checking that each message the simulated party sent is correct according to the BGW protocol.

- The only other check that the verifiers need to do is to ensure that the their transcripts are consistent—that at each step the message that party i sent to party j in the transcript is the same message that party j received from i over their point-to-point channel.

  To do this, each pair of verifiers takes a hash of the messages sent over their point-to-point channel.

  The verifiers exchange their hashes and check that they match.

  This can be a collision-resistant hash function or even a simpler information-theoretic one.

- If the simulated output of the BGW protocol has $C(z) = 1$, then the verifiers accept. Otherwise they reject.

Efficiency. The prover must send each verifier a number of $\mathbb{Z}_p$ elements proportional to $|C|$. So the circuit C must be simple/small for efficiency. The verifiers only exchange on $\mathbb{Z}_p$ element for each client.

To sketch the security argument:

- **Completeness.** Follows from BGW.
- **Soundness.** Follows again from BGW. For the prover to cheat, there must be at least one message in one transcript

that deviates from what the party in BGW would have sent. The verifiers' checks will catch this.

- **Zero knowledge.** Follows again from BGW. The transcript leaks nothing to a malicious coalition of verifiers. The verifiers only exchange a single message with each other to check the transcripts. The only information that a malicious subset of verifiers learns from the honest parties' messages is the hash of each set of messages sent over the point-to-point channels. The malicious verifiers already have these messages, so it is possible to simulate these messages without the honest parties' inputs.

Outlook

As long as we confine

References