Privacy Enhancing Technologies FS2025
Lecture 35 – Distributed Private Computation

Florian Tramèr

AGENDA

1. Can we have cryptographic "secrecy" and privacy?
2. Private Aggregation and Applications
3. Distributed noise generation
4. (Optional) PRIO: ZK proofs on secret-shared data

Recap

We're reaching the end of this course. In the first half of the course, we focused on cryptographic tools for "privacy" (defined as "leaking no information other than the output"). In the second half, we considered what can leak from the output themselves, and looked at statistical tools to protect this kind of "privacy" (defined as "leak not much more information than if you hadn't participated").

To conclude the course, we'll now see how to combine these tools to get both forms of privacy, without requiring trust in any single entity. We'll cover a very nice protocol for private aggregation, PRIO [CGB17], which combines ideas from many of our previous lectures: secret sharing, non-colluding servers, SNARGs, (some variants also use Distributed Point Functions), Differential Privacy, etc. It is also a nice recent "success story" for cryptographic PETs, which had led to large-scale real-world deployments by Mozilla, Apple, Google, Cloudflare, and others, and is currently being standardized by the IETF.[1]

Finally, I think this result is also a super neat marriage of theory and practice. An early version of the protocol we'll discuss was proposed in [CGB17], with a focus on practical efficiency. But it turned out that the protocol could be significantly improved (both asymptotically and in practice) by drawing connections to the broader theory of SNARGs [BBCG+19].

1 Simultaneously Solving *How* and *What*

The guarantees offered by cryptographic approaches to privacy and differential privacy are quite different. With differential privacy, we get a guarantee that is statistical and that protects against *any* inference an attacker could make that is specific to a single individual. This is stronger than the guarantees offered by cryptography, which are computational and don't protect against inference attacks based on protocol outputs. The flip side is that the best DP protocols require a trusted curator, and must allow for (at least) constant leakage rather than negligibly small leakage. In contrast, cryptography allows to securely compute *any* efficiently computable function, with negligible leakage beyond the function's output.

[1] https://datatracker.ietf.org/doc/draft-ietf-ppm-dap/

This suggests a very natural way to combine the two: first decide *what* (differentially private) function the parties want to compute, and then use cryptography to figure out *how* to compute this function securely [BNO08]. (one way to think about this is that our *ideal functionality* is now a differentially private mechanism, and we'll use MPC techniques to implement it in the real world.)

DP models. So suppose we have some function f that n parties want to compute, while satisfying some differential privacy guarantee and without trusting any central entity with all their inputs.

There are many ways to do this, with different trade-offs between privacy, utility, and trust requirements. To illustrate the different approaches, we'll consider our canonical problem of computing counting queries over the users' inputs.

- **The central model:** Of course, the baseline is to just trust a central curator with all the inputs... Then we can get the optimal error $\tilde{O}(\sqrt{k}/\varepsilon\sqrt{n})$ using the Gaussian mechanism.
- **The local model:** If the parties don't want to trust anyone (and not even cryptography), they can each add noise to their input and share them in the clear with each other to compute the function (e.g., using the Randomized Response mechanism).

  The downside is that local DP has a large impact on utility (since each party has to add enough noise to nearly drown the signal). Specifically, for k counting queries, the amount of noise is $\tilde{O}(1/\varepsilon\sqrt{n})$.
- **The shuffle model:** This is an intermediate model between the local and central models, that has received a lot of attention recently [BEM+17, EFM+19, CSU+19]. While we won't cover it in detail, the idea is that the parties send their (noised) data to a trusted *suffler*, whose just randomly permutes the data before sending it to the central curator.

  Shuffling can be shown to provide a form of privacy amplification (as with subsampling). For counting queries, this allows to match the error of the central model (up to logarithmic factors). Unfortunately, for some queries (e.g., selection queries), the error of the shuffle model is much larger than the error of the central model.

  But it seems we have just replaced one trusted entity with another. The crux is that the shuffling is a very simple mechanism, that is independent of the function being computed. There are a number of custom cryptographic protocols for anonymous communication (e.g., Mix-nets, Tor, etc.) that can be used to securely implement the shuffler.
- **The MPC model:** If we want to match the utility of the central model for all functions without trusting a central curator, we can use a generic secure MPC protocol to compute the differentially private mechanism.

  This is the most general solution, but it remains rather inefficient in practice and thus seldomly used (except for relatively simple functions).

2 Private Aggregation

We'll now build a protocol that combines both statistical privacy (i.e., DP) and cryptographic privacy (i.e., secure function evaluation). We'll focus on a very simple but ubiquitous task. There is a large number n of parties, each holding a vector x_i of d values:

$$x_1, x_2, \dots, x_n \in \mathbb{F}_p^d .$$

We want to collect the aggregate value

$$\sum_{i=1}^{n} x_i \in \mathbb{F}_p^d .$$

We can consider other *linear* functions of values released by the users. This allows, for instance, to compute variances, regression coefficients, and even approximations to some non-linear functions such as "heavy hitters" (i.e., the most frequent values in a dataset).

2.1 Applications

Such data aggregation is useful for many applications, typically for collecting statistics over a large number of (often computationally weak) users.

- **Telemetry:** A company may want to collect statistics over the usage of their product, e.g., which failures occurred most often, or which features are most popular. Of course, collecting such data directly from the users would be very bad for privacy.
- **Public health monitoring:** During the COVID-19 pandemic, Apple and Google deployed exposure notification apps, which notify users if they have been in contact with an infected person. Due to privacy concerns, these systems were designed so that Apple, Google, and other third parties don't learn if users were shown an exposure notification, or who they came in contact with. This information is kept strictly on device. Yet, to track the spread of the virus, it would be useful for health authorities to obtain some global statistics, e.g., the number of people who came in contact with an infected person in a given region.
- **Federated Learning:** In Federated Learning, clients hold a machine learning model and want to aggregate the updates (i.e., gradients) from all clients to get a new model. This is just a linear function of the users' (very high-dimensional) values.

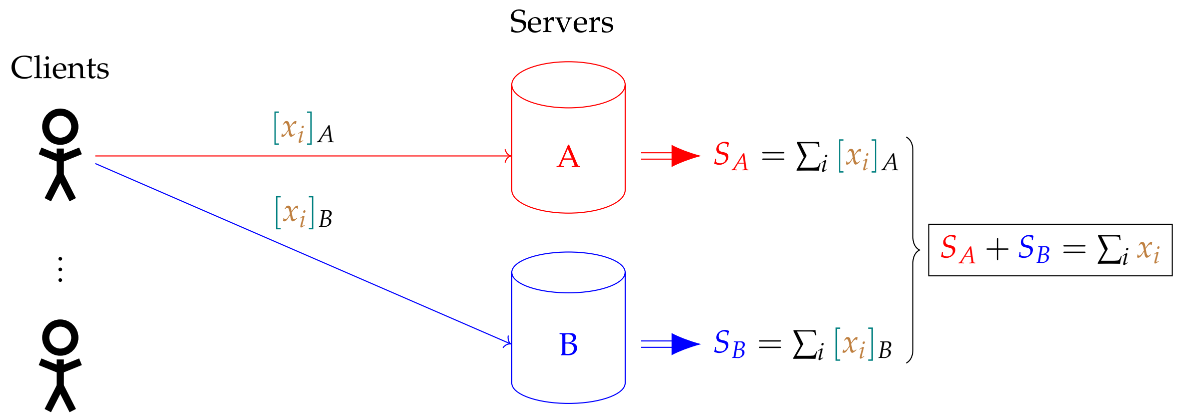
2.2 Approaches

Given such an aggregation task, there are different approaches to compute it securely.

- **Generic MPC:** We could of course compute the linear function directly with a full-blown (maliciously secure) MPC protocol between all clients. But this is rather inefficient, both in computation and communication for all the clients (the communication costs of MPC protocols are typically quadratic in the number of parties).
- **Server-assisted MPC:** Another approach is to have a central server that coordinates all clients, and does the heavy computation. One solution is to use linear homomorphic encryption, but this requires all clients to jointly generate a secret-shared key. Another approach relies on having clients agree on pairwise random masks to conceal their inputs (which cancel out when adding them together) [BIK+17], but this entails a large communication overhead.
- **Distributed trust:** A pragmatic approach, which has become very popular in recent years, is to rely on *multiple* servers to assist the clients in the computation—under the assumption that the servers are not colluding.

  We've already seen examples of such protocols in the previous lectures, e.g., the 2-server PIR protocol. These protocols are typically significantly more efficient than protocols with a single server (e.g., by requiring only symmetric-key cryptography instead of expensive public-key cryptography). The main practical challenge is to find servers that are willing to run the protocol, and that can be trusted not to collude.

We'll now focus on the last approach, which is the most efficient one in practice. We'll assume we have two *non-colluding* servers, although the ideas generalize to more than two servers. A simple private aggregation protocol would then look like this:



That is, each party additively secret-shares their input between both servers, and then the servers add all their shares together before revealing their share of the sum.

Properties of this simple protocol:

- **Correctness:** If all parties act honestly, we get the correct answer.
- **Privacy:** If one server is honest, the protocol leaks nothing beyond the sum $\sum_i x_i$ (we'll get back to DP later).
- **Efficiency:** Clients send a single message to both servers. The servers exchange a single field element.

2.3 The Integrity Problem

The above protocol provides perfect privacy! But this comes at a significant cost: any user can completely disrupt the *integrity* of the protocol's computation.

Indeed, in the applications above, the parties' inputs are not *arbitrary* field elements. (if they were, would learning their sum be useful?) Rather, they are typically bounded in some range. For example, for telemetry, each client's data might be a vector of bits.

But our simple protocol above puts no limit on a client's inputs. This means that a single malicious user can arbitrarily disrupt the protocol's result...[2] That is, an evil user can just set $[x_i]_A + [x_i]_B$ to be any arbitrary vector in $\mathbb{F}_p^d$. Then, the protocol outputs

$$\sum_i x_i + \Delta \,,$$

where $\Delta \in \mathbb{F}_p^d$ is an arbitrary shift. We thus need to be able to constrain user inputs, such that we compute

$$\sum_i x_i \quad \text{for } x_i \in \mathcal{X} \subset \mathbb{F}_p^d \,.$$

For example, for telemetry, we might have $\mathcal{X} = \{0,1\}^d$.[3]

[2]Note that other approaches based on MPC would suffer from the same issue.
[3]A malicious client can still send an arbitrary element in $\mathcal{X}$. The protocol cannot prevent this.

You might recognize this problem from your midterm! In that problem, we covered a special case of this protocol, where we want to show that a secret-shared value is a unit vector. We describe the full protocol, called PRIO [CGB17], in Appendix A (for your reading pleasure, or if you feel nostalgic about the midterm!).

3 Distributed Noise Generation

What's missing from our protocol is Differential Privacy! We now have a secure and robust way to compute sums (or other linear functions) of secret-shared data, but we still need to add noise to the output to get differential privacy.

The PRIO protocol in Appendix A doesn't specify how to do this, but there is a large body of work on this problem, known as *distributed noise generation* [DKM+06]. The main idea is to have the aggregators (e.g., the two servers in our PRIO protocol) generate shares of the appropriate noise (we'll focus on Gaussian noise here), and add them to their output shares before revealing them.

There are different possible approaches, depending on the type of noise we want to add and the trust model we are in.

- **The non-malicious case:** Suppose the noise shares are generated by m parties, and we assume at least t of them are honest (the others might generate shares of zero noise to leak more information). Then, each party can create shares of Gaussian noise of mean 0 and variance σ^2/t.[4] This guarantees that the secret-shared noise has variance at least σ^2.

  However, we might also end up with too much noise, of variance up to $m/t \cdot \sigma^2$ if all parties participate honestly. Another issue is that a single malicious noise generator can completely disrupt the protocol by creating shares of arbitrarily large noise (note that the PRIO protocol doesn't guard against malicious aggregators).

- **The malicious case:** To guard against malicious noise generators, we could use a generic MPC protocol for the entire noise generation process. Alternatively, we could have the noise generators prove, in ZK, that their noise shares are within the correct range (e.g., using a similar approach as in PRIO).

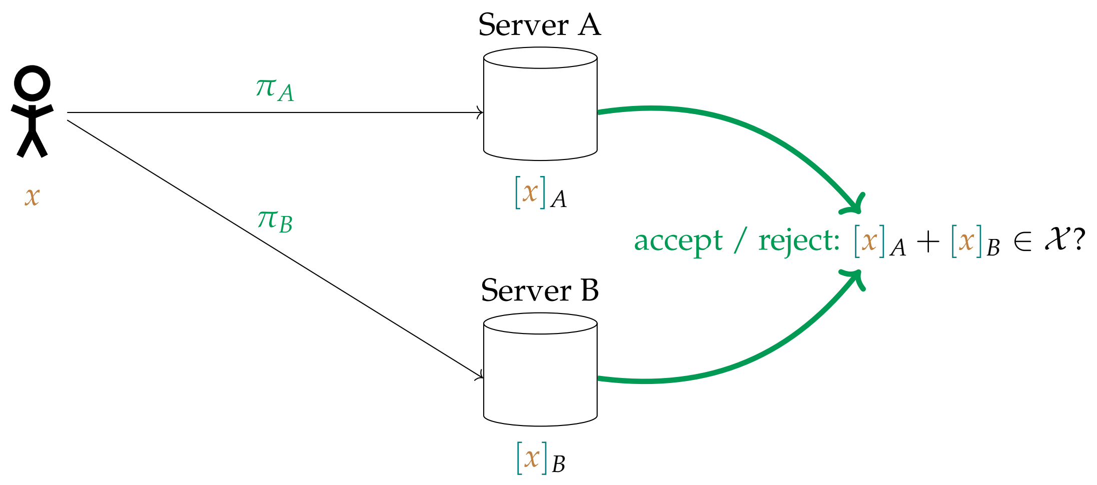
  The first maliciously-secure protocol due to Dwork et al. [DKM+06] roughly works as follows. We start with the observation that a Gaussian distribution of mean 0 and variance σ^2 can be approximated by a suitable binomial distribution. To generate shares of this distribution, the parties generate many shares of uniformly random bits $[b_i]$, where $b_i \in \{0, 1\}$, and prove to each other that these are indeed shares of bits (there are specialized, efficient ZK proofs for this).

  These bits could still be adversarially biased though. One solution for this is for the parties to open some bits publicly, and use these bits to generate a public stream of uniformly random bits $c_i \in \{0, 1\}$ (e.g., using a PRG). The parties can then use these public bits to *bias* the secret-shared bits, to get a stream of secret-shared bits $[b_i \oplus c_i]$ that are uniformly random.

[4] Since we are operating over a finite field, we can't directly use the Gaussian distribution. Instead, we can use the discrete Gaussian distribution, which is a distribution over the integers that is close to the Gaussian distribution [CKS20].

A PRIO: ZK Proofs on Secret-Shared Data

Recall the integrity problem we introduced in Section 2.3. We're going to fix this issue with ZK proofs! The client will prove to both servers that their input is well-formed. If the servers trust the proof, they will add the client's shares to their sum, and otherwise reject them.



This proof system differs slightly from ones we've seen before, in that the statement x is not directly visible to the servers. Instead, they can only access the statement in secret-shared form. Such proof systems are called *zero-knowledge proofs over secret-shared data* [BBCG+19].

The properties we'll want are analagous to the ones we've seen before:

- **Completeness:** If the client is honest, the servers accept the proof.
- **Soundness:** If $x \notin \mathcal{X}$, a malicious client can convince an honest server with only negligible probability.
- **Zero-knowledge:** The servers learn nothing about x, except that it is in $\mathcal{X}$ (unless they collude of course, in which case all privacy is lost).

A.1 Polynomials (Again)

We'll now describe a very cute ZK proof system for secret-shared data, due to [CGB17] and [BBCG+19]. The protocol can be seen as an (extremely) simplified version of the PLONK IOP we saw before. The reason we will get something much simpler than PLONK is due to two factors: (1) there are two non-colluding verifiers, which means we can secret-share data instead of using commitments; (2) the proofs won't actually be succinct.

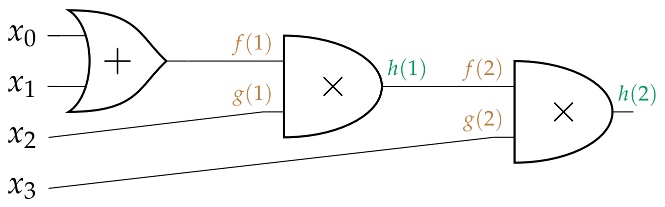
Let $\mathcal{C} : \mathbb{F}_p^d \to \{0,1\}$ be an arithmetic circuit for testing the validity of an input. That is, $\mathcal{C}$ outputs 0 if and only if $x \in \mathcal{X}$. We thus want a proof that $\mathcal{C}(x) = 0$.

The client (the prover) will start by running the circuit themselves. For each of the T multiplication gates in the circuit, let (u_t, v_t) be the inputs to that gate. Sample u_0 and v_0 randomly from $\mathbb{F}_p$ and define two degree-T polynomials f and g such that

$$f(t) = u_t \quad \text{and} \quad g(t) = v_t \quad \text{for } t \in \{0, \dots, T\} .$$

Then, define the polynomial $h := f \cdot g$. Note that h is a polynomial of degree at most $2T$, which encodes the *outputs* of each multiplication gate in the circuit, i.e.,

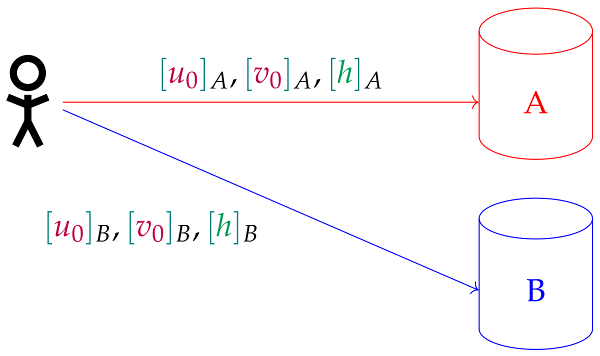
$$h(t) = f(t) \cdot g(t) = u_t \cdot v_t \quad \text{for } t \in \{1, \dots, T\} .$$



The client then splits u_0, v_0 and h into shares:[5]

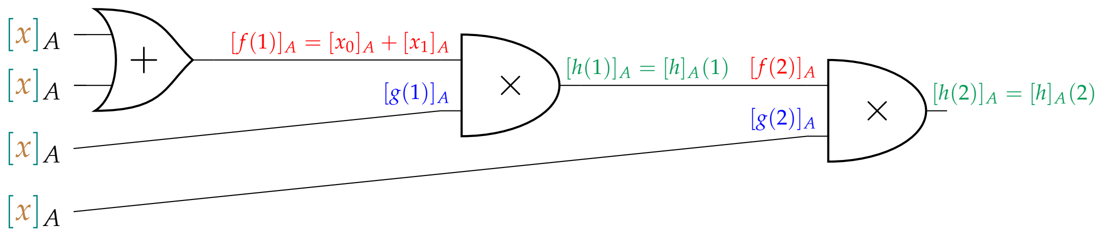
$$u_0 = [u_0]_A + [u_0]_B \quad \text{and} \quad v_0 = [v_0]_A + [v_0]_B \quad \text{and} \quad h = [h]_A + [h]_B$$

Finally, the client sends the corresponding shares to the two servers:



The nice observation here is that the servers can reconstruct shares of the polynomials f and g from all the shares they have received, without any communication. First, let's see that the servers can reconstruct shares of the values of all wires in the circuit:

- Server i has a share of each of the input values $[x]_i$
- Server i has a share of each output of a multiplication gate (i.e., $[h]_i(t)$ for $t \in \{1, \ldots, T\}$)
- The servers can calculate shares of all other wires with linear operations on these shares.



Then, given shares of $[f(i)]$ and $[g(i)]$ for $i \in \{0, 1, \ldots, T\}$, the servers can interpolate polynomial shares $[f]_A, [f]_B$ and $[g]_A, [g]_B$. If all parties acted honestly, the servers end up with shares of polynomials f, g, and h such that $f \cdot g = h$.

It is not too hard to see that if the malicious client sends shares of a polynomial $\hat{h}$ that does *not* interpolate the outputs of all multiplication gates in $C(x)$, then the servers will reconstruct shares of polynomials $\hat{f}$ and $\hat{g}$ for which $\hat{f} \cdot \hat{g} \neq \hat{h}$.

So what's left to do for the servers? They need to check that:

1. They hold shares of polynomials f, g and h such that $f \cdot g = h$.
2. $h(T) = 0$.

[5]To secret share a polynomial, we write it out as $h(X) = \beta_0 + \beta_1 X + \beta_2 X^2 + \ldots$ and then secret-share the vector of coefficients $[\beta_0], [\beta_1], [\beta_2], \ldots$.

Let's start with point (2). Here, each server just evaluates $[h]_i(T)$ locally and shares the value with the other server. Then, they check that $[h]_A(T) + [h]_B(T) = h(T) \overset{?}{=} 0$.

For (1), recall the polynomial equality test we used for PLONK: the servers agree on a random value $r \leftarrow^{\$} \mathbb{F}_p$ and then check that $f(r) \cdot g(r) \overset{?}{=} h(r)$. Again, the servers can compute shares of $f(r)$, $g(r)$ and $h(r)$ locally, and then reveal the shares to check the equality. No need for any fancy polynomial commitment scheme here!

Zero-knowledge. If at least one server is honest, the protocol provides *perfect* zero-knowledge! That is, even a malicious server with unbounded compute power learns nothing about the input x, except that it is in $\mathcal{X}$.

Informally, because we sampled the y-intercept of the polynomials f and g (i.e., u_0 and v_0) at random, the values $f(r)$ and $g(r)$ observed by the servers are uniform in $\mathbb{F}_p$ and independent of the adversary's view. The other values seen by the adversary are additive secret shares (which are information-theoretically hiding), and the value $h(T) = 0$ which reveals nothing more than the fact that $x \in \mathcal{X}$.

Formalizing this argument takes a bit more work. One subtlety is that zero-knowledge only holds if the challenge $r \notin \{1, \ldots, T\}$ (otherwise, releasing $f(r)$ and $g(r)$ leaks information about the input x). If at least one server is honest, we can ensure this doesn't happen.

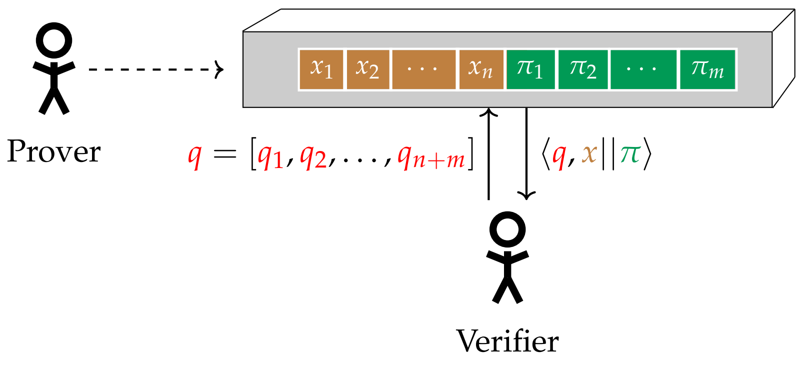
Complexity. What's the complexity of this proof system? The proof π is of size $O(T)$, which is not succinct in general (i.e., we can have $T = O(|\mathcal{C}|)$). Getting around this is possible with more involved (multi-round) protocols.

The server's computation complexity to verify the proof is also $O(T)$. But what's really nice is that the servers only need to exchange $O(1)$ field elements per proof!

A.2 Generalization: (Fully) Linear PCPs

The protocol above is a case of a more general class of proof systems: fully linear PCPs [BCI+13, BBCG+19]. Recall that in the standard PCP theorem, we show that given a statement x and proof π, a verifier can check the proof by querying it at three random positions.

There are other PCPs where the verifier instead makes only *linear* queries to the proof string, concatenated with the statement. That is, given a statement $x \in \mathbb{F}_p^n$ and proof $\pi \in \mathbb{F}_p^m$, the verifier can ask queries of the form $q \in \mathbb{F}_p^{n+m}$ and receive the inner product $\langle q, x||\pi \rangle \in \mathbb{F}_p$. They then accept or reject the proof based solely on such inner products.



Linear PCPs are neat for a number of reasons. First, they can be implemented much more efficiently in practice compared to regular PCPs. In fact, our polynomial check above is a linear PCP in disguise! The statement is the input $x \in \mathbb{F}_p^d$, and the proof π consists of the coefficients of the polynomials f, g and h (we actually only need the first coefficients of f and g as we saw above). Then, the linear PCP verifier checks that $f \cdot g = h$ and $h(T) = 0$, with requires only linear operations over x and the proof. Overall, we can represent the verifier checks as four linear queries q_i to a proof of size $O(T)$ where T is the number of multiplication gates in the validation circuit.

The second reason why linear PCPs are neat is that, in cryptography, there are many ways to "hide" data such that linear operations over the hidden data are possible (e.g., linearly homomorphic commitments or encryption, and additive secret sharing)!

So, given any (zero-knowledge) linear PCP, we immediately get a (zero-knowledge) proof over secret-shared data! The prover sends additive shares of x and π to the verifiers, who evaluate the same random linear PCP queries locally.[6] They then reveal their shares of the outputs to reconstruct the inner products $\langle q_i, x||\pi \rangle$ and run the linear PCP verifier.

This view is actually very useful, because there are many different linear PCPs in the literature. By adding these to our private aggregation protocol, we can get protocols with different tradeoffs in communication and computation, for different types of circuits $\mathcal{C}$. For example, if the verification circuit $\mathcal{C}$ has some special structure (e.g., a sub-circuit that is replicated many times), then it is possible to build linear PCPs (and thus a private statistics protocol) with better communication complexity. Moreover, if we allow for interaction between provers and verifiers (i.e., a linear *IOP* [BCI+13]), we can get succinct proofs for arbitrary circuits.

References

[BBCG+19] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In *Annual International Cryptology Conference*, pages 67–97. Springer, 2019.

[BCI+13] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Omer Paneth, and Rafail Ostrovsky. Succinct non-interactive arguments via linear interactive proofs. In *Theory of Cryptography: 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, pages 315–333. Springer, 2013.

[BEM+17] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proceedings of the 26th symposium on operating systems principles*, pages 441–459, 2017.

[BIK+17] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.

[BNO08] Amos Beimel, Kobbi Nissim, and Eran Omri. Distributed private data analysis: Simultaneously solving how and what. In *Advances in Cryptology–CRYPTO 2008: 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings 28*, pages 451–468. Springer, 2008.

[6]We assume here that the servers have some shared randomness, e.g., a shared key for a pseudorandom number generator (PRG).

[CGB17] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX symposium on networked systems design and implementation (NSDI 17)*, pages 259–282, 2017.

[CKS20] Clément L Canonne, Gautam Kamath, and Thomas Steinke. The discrete gaussian for differential privacy. *Advances in Neural Information Processing Systems*, 33:15676–15688, 2020.

[CSU$^+$19] Albert Cheu, Adam Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. Distributed differential privacy via shuffling. In *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I 38*, pages 375–403. Springer, 2019.

[DKM$^+$06] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *Advances in Cryptology-EUROCRYPT 2006: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28-June 1, 2006. Proceedings 25*, pages 486–503. Springer, 2006.

[EFM$^+$19] Úlfar Erlingsson, Vitaly Feldman, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Abhradeep Thakurta. Amplification by shuffling: From local to central differential privacy via anonymity. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2468–2479. SIAM, 2019.