Constructing PCPs, Arguments

CS6230: Topics in Information Security
Lecture 10 (Oct 19, 2021)
Lecture and notes: Prashant Nalini Vasudevan

In the previous lecture, we saw the application of PCPs to proving the hardness of approximating optimisation problems, and started looking at the construction of a PCP for the problem of solving a system of linear equations. Today, we will wrap up this construction, though we will leave the proof of an important part of it to the next problem set. The problem set will also cover the extension of this PCP to solving systems of quadratic equations. This is an NP-complete problem, and thus the PCP will lead to a weak version of the PCP theorem – one with constant queries, but polynomial randomness complexity and an exponentially long proof.

Later in this lecture, we will see a different application of PCPs that has led to a lot of exciting developments in recent years – constructing efficient computationally sound interactive proofs.

1 A Simple PCP for ML

Recall that the problem we wished to construct a PCP for was the Multivariate Linear (ML) problem – given matrix $A \in \mathbb{F}_2^{m \times n}$ and vector $b \in \mathbb{F}_2^m$, determine whether there exists a solution to the system of linear equations $Ax = b$. The PCP proof for an instance (A, b) that had a solution $u \in \mathbb{F}_2^n$ consisted of the encoding H_u of u, which was a string of length 2^n, where for any $a \in \mathbb{F}_2^n$, $H_u[a]$ was the inner product $\langle a, u \rangle$. Given a promise that a candidate proof π had the form H_v for some vector v, the verifier's strategy was to pick a random $a \in \mathbb{F}_2^n$, and check that $\pi[a]$ was indeed the inner product of a with some vector u that satisfied $Au = b$. For this, we used the fact that two linear functions over $\mathbb{F}_2$ agree on exactly half of all inputs.

1.1 Error-Correcting Codes

The set of encodings $\{H_u\}_{u \in \mathbb{F}_2^n}$ we used above is what is known as an *Error-Correcting Code* (ECC). We have seen examples of such codes multiple times so far, but we never really referred to them by name. An error-correcting code is an encoding of strings such that any two strings that differ even in one location are mapped to strings that are very different. These encodings are one of the most pervasive objects in computer science, and have found widespread use in data storage, communication, cryptography, etc.. They are fascinating, but unfortunately, we will not have time to cover them in much detail here.

For historical reasons, such an encoding $E : \mathbb{F}^n \to \mathbb{F}^t$ is said to map a *message* x to a *codeword* $E(x)$. Generally in ECCs, two parameters are of importance:

- The *rate*, which is the ratio of the size of the message to the size of the codeword – n/t. Generally, in a good code, this should be close to 1. This would mean that the encoding does not have much overhead.
- The *distance*, which is the minimum (relative) Hamming distance between the encodings of any two distinct messages. In a good code, this should also be close to 1. This would mean that, even if some large fraction of bits of the codeword were adversarially changed, it would still be possible to recover what the message it encoded was.

The specific code we used above – where a codeword consists of the inner product of the message with all the vectors in the space – is called the Hadamard code. This code has very bad rate ($n/2^n$), but very good distance ($1/2$). It also has two useful properties – it is *locally testable* and *locally correctable*. Local testing enables the first part of the verification task as noted earlier. Local correction helps deal with the fact that local testing is not perfect. In other contexts, local correction also helps get worst-case to average-case reductions between problems.

The first step in getting better PCP constructions is to use a code that has both of these properties, but also has much better rate than the Hadamard code. This would lead to a shorter proof string and often less randomness complexity. One such code is the Reed-Solomon code, which consists of evaluations of a low-degree univariate polynomial, whose co-efficients are specified by the message. This is much like how the linear function in the Hadamard code was specified. Another is the Reed-Muller code, which

is the same but with low-degree multivariate polynomials (much like how the Hadamard code involved *linear* multivariate polynomials). These codes were implicit in many of the interactive protocols we saw earlier in the class.

1.2 Linearity Testing

The first task of the verifier, which we skipped over earlier, was, given oracle access to the proof string π, to check that it is the Hadamard encoding H_v of some string $v \in \mathbb{F}_2^n$. If it could look at the entire string π, it could have done this perfectly, but since it has only a small number of queries it can make, the verifier can only ensure that π is close to such an encoding. In general, the task of testing whether a given string is a valid codeword of some code by making a small number of queries to it is called *local testing*. The study of local testability of codes started with early constructions of PCPs, but has since gathered interest in its own right, and a breakthrough in the construction of locally testable codes with good rate and distance was just announced two weeks ago [DEL+21].

In the particular case of the Hadamard code, the task of local testing comes down to testing whether a given function is linear. This is because, as noted earlier, any valid codeword H_v of the code can be thought of a the truth table of a linear function $h_v(a) = \langle v, a \rangle$. Then, given an oracle for a function $h : \mathbb{F}_2^n \to \mathbb{F}_2$, the task is to determine whether the function is linear by making a small number of queries to it. This is known as *linearity testing*. Again, it is not possible to check whether the function is perfectly linear, so the verifier is only required to reject if the function is somewhat far from being linear.

Lemma 1.1. *There is a linearity tester L that, given oracle access to a function $h : \mathbb{F}_2^n \to \mathbb{F}_2$, makes a constant number of queries to it, runs in $O(n)$ time, and has the following properties:*

1. *If h is linear, L^h accepts with probability* 1
2. *Suppose that, for any linear function $\ell : \mathbb{F}_2^n \to \mathbb{F}_2$, the outputs of h and ℓ differ in more than a* 0.01 *fraction of inputs. Then L^h rejects with probability at least* 0.9.

The test itself turns out to be quite simple – pick random inputs $r, s \in \mathbb{F}_2^n$, and check whether $h(r) \oplus h(s) = h(r \oplus s)$. The analysis is a bit involved, and we will see it in the next problem set. For now, we will see how to complete the PCP verifier if this test were used to realise the first part of the verification strategy – to check that π is close to a valid codeword H_u.

1.3 Putting things together

With the linearity tester L from Lemma 1.1, we can now describe the entire PCP verifier as follows given input (A, b) and access to a proof π:

1. Run L^π and reject if it rejects
2. Sample random $r \leftarrow \mathbb{F}_2^m$, and verify that $\pi[r^T A] = \langle r, b \rangle$

The arguments from the previous lecture show that this verifier would work if L tested for perfect linearity. It is not much harder to show that it continues to work even if L only tests approximate linearity in the sense of Lemma 1.1. This is because the query $r^T A$ that the verifier makes to π is uniformly distributed over all indices of π (under the assumption that A is full rank). So even if π were actually 0.01-far from any linear function and L is not able to catch it, this 0.01 fraction of bad queries can be absorbed into the soundness error that the PCP verifier is allowed. Overall, the verifier uses $O(1)$ queries, and $O(m)$ bits of randomness.

2 PCPs and Arguments

The other major application of PCPs is in constructing efficient interactive proofs with computational soundness properties. These proofs, called *interactive arguments*, are interactive proofs where the prover is required to be computationally efficient (perhaps given some witness), and the soundness guarantee is required only against cheating provers that are also computationally efficient.

Definition 2.1. An interactive protocol (P, V), where both P and V run in probabilistic polynomial time (in the length of the input), is said to be an *interactive argument* for a language L if:

- **Completeness:** For every $x \in L$, there exists a $w \in \{0,1\}^*$ such that when (P, V) is run with x as input to V and (x, w) as input to P, and security parameter λ, V accepts at the end with probability at least $1 - \text{negl}(\lambda)$.
- **Soundness:** For every $x \notin L$, and any prover strategy P^*, when (P^*, V) is run with x as input to V and security parameter λ, V accepts at the end with probability at most $\text{negl}(\lambda)$.

Exercise 1. *Show that interactive arguments can only exist for languages in* NP.

Of course, any NP proof system itself is an argument, as the prover could simply send the proof over, but we are typically interested in arguments that have much smaller communication and verifier running time than this.

Arguments are incomparable to IPs in general because they ask for the honest prover to be computationally efficient (perhaps given a witness), which is a stronger requirement than in IP, but they only ask for soundness to hold against efficient cheating provers, which is a weaker requirement. In practical applications, computational efficiency of the prover is important in any case, and so practical arguments are often easier to construct than practical IPs. Further, security against computationally efficient adversaries is also often sufficient in applications. For this reason, when it comes to practice, arguments are right kind of interactive proofs to consider, though constructions of arguments often involve constructions of information-theoretically sound proof systems as a first step.

Arguments from PCPs. An example of such a construction is Kilian's argument for languages in NP using PCPs [Kil92]. In order to construct an argument system for a language $L \in \mathsf{NP}$, this approach starts with a PCP for L, and specifies how the proof oracle can be implemented securely using a prover who knows the witness. That is, the verifier V in this argument system runs the PCP verifier V_{PCP}, and whenever V_{PCP} makes a PCP oracle query, V sends the query to the prover P, who should respond with the corresponding bit of the PCP proof π. Using existing polynomial-length PCPs, the PCP proof π for any input $x \in L$ can be computed in polynomial-time by the prover given a witness w for x, leading to an argument system with perfect completeness.

Achieving soundness in this approach, however, is more complicated, and makes use of the fact that we only require security against polynomial-time cheating prover strategies. The salient property of the PCP model here is that the proof π encoded by the oracle is fixed ahead of time, before the verifier makes the queries, and so its response to any query cannot change depending on what other queries are made. If the verifier V simply forwarded V_{PCP}'s queries to a cheating prover P^*, however, P^* can decide what response to send to one query based on what some other query was. This breaks the PCP abstraction, and we cannot use the PCP's soundness guarantee any more.

Kilian's solution to this was to have the prover *commit* to the proof π before the start of the protocol, and then whenever it answers a query of V_{PCP} that was forwarded by V, it would have to prove, through the commitment scheme, that it is answering the query in a way that is consistent with the initial commitment it made. This turns out to be sufficient to realise the PCP model accurately, and make use of the PCP's soundness. We will cover his construction at a high level, introducing the necessary techniques, but not delving much into details of the proofs.

2.1 Merkle Hashing and Commitments

We have encountered cryptographic commitment schemes earlier in the course, in the construction of CZK proofs for NP. Recall that this is an interactive protocol with two phases – commitment and decommitment. In the commitment phase, the sender S and receiver R interact, with the transcript of this interaction implicitly committing S to some message m. In the decommitment phase, S reveals to R the message m and the random string it used that correctly explains the transcript from the commitment phase. There, we used a commitment scheme that was statistically binding and computationally hiding – we wanted the binding property to protect against the computationally unbounded prover, and the hiding property to protect against the efficient verifier.

Here, as we only have to deal with computationally efficient provers, we will use a commitment scheme that has the strength of these properties swapped – it will be *computationally binding* and *statistically*

hiding. In addition, it will have the property that commitments to even long messages are very short, and also that any bit of the committed message can be decommitted much more efficiently than decommitting the entire message. This last property is called *local opening*. We will not write down a definition of such a commitment scheme, but will rather just see a construction of one – *Merkle Hashing*.

Collision Resistance. This commitment scheme makes use of another fundamental cryptographic primitive – a family of *Collision-Resistant Hash Functions* (CRHF). These are families of functions where it is hard to find two inputs that are mapped to the same output.

Definition 2.2. For some functions $n, m : \mathbb{N} \to \mathbb{N}$, consider a family of functions $H = \{H_\lambda\}_{\lambda \in \mathbb{N}}$, where $H_\lambda = \left\{h : \{0,1\}^{m(\lambda)} \to \{0,1\}^{n(\lambda)}\right\}$. H is said to be a family of *collision resistant hash functions* if:

- **Efficiency:** For any λ, a random $h \in H_\lambda$ can be sampled in $\mathsf{poly}(\lambda)$ time. Further, any $h \in H_\lambda$ can be evaluated on any input in $\mathsf{poly}(\lambda)$ time.
- **Shrinkage:** For all large enough λ, $n(\lambda) \leq m(\lambda)/2$ – that is, the functions h are shrinking.
- **Collision Resistance:** For any PPT algorithm A and all large enough λ, when $h \leftarrow H_\lambda$ and $(x, x') \leftarrow A(1^\lambda, h)$, the probability that $x \neq x'$ and $h(x) = h(x')$ is $\mathrm{negl}(\lambda)$.

For the rest of our discussion, we will fix some large security parameter λ. A CRHF, then, is a family of functions $H = \{h : \{0,1\}^m \to \{0,1\}^n\}$ such that, say, $m = 2n$. That is, its output is much smaller than its input, meaning there are many pairs of distinct inputs (x, x') such that $h(x) = h(x')$ – such inputs are said to "collide". And yet, it is hard to find such colliding inputs efficiently. Note that we need the family to contain super-polynomially many h's, as otherwise an adversary could just memorise pairs of collisions for every possible h. CRHFs are a basic cryptographic primitive that are widely used in practice, though the security of most implementations tends to be heuristic.

Merkle Trees. Seen one way, a Merkle Tree is a way to start with a hash function h that maps $2n$ bits to n bits, and get another h^k that maps $2^k \cdot n$ bits to m bits for some $k > 1$. This is done by first splitting up an input of length $2^k \cdot n$ into 2^k blocks of length n each, and repeatedly applying it on pairs of blocks in a tree structure, until only one block remains. More precisely, given $h : \{0,1\}^{2n} \to \{0,1\}^n$, the function h^k is computed as follows given an input $x \in \{0,1\}^{2^k n}$:

1. Split x into strings $(x_{0,0}, x_{0,1}, \ldots, x_{0,2^k-1})$, each of length n.
2. For i from 1 to k:
   - For j from 0 to 2^{k-i}, compute $x_{i,j} \leftarrow h(x_{i-1,2j}, x_{i-1,2j+1})$
3. Output $x_{k,0}$

This process may be thought of as populating the nodes of a binary tree of depth k where the leaves are labelled by the input, and the internal nodes by the hash of their children under h, with the label of the root being the output of h^k. Given a family of hash functions $H = \left\{h : \{0,1\}^{2n} \to \{0,1\}^n\right\}$, consider for any $k \in \mathbb{N}$ the family $H^k = \left\{h^k : \{0,1\}^{2^k n} \to \{0,1\}^n\right\}$, which contains h^k for each $h \in H$.

Claim 2.1. *For any $k = O(\log n)$, if H is a CRHF family, then so is H^k (though with some loss in the security level).*

Proof Sketch. First, if H can be efficiently samplable and evaluated, then so can H^k as long as $k = O(\log n)$, as the number of evaluations of h required to evaluate h^k is 2^k.

Suppose, for an $h^k \in H^k$, an adversary is able to find a pair of distinct inputs $x, x' \in \{0,1\}^{2^k n}$ such that $h^k(x) = h^k(x')$. We will show how to extract from this a collision for the corresponding $h \in H$. Split up x (and similarly x') into $(x_{0,0}, \ldots, x_{0,2^k-1})$. There has to exist some j such that $x_{0,j} \neq x'_{0,j}$. And, by the fact that $h^k(x) = h^k(x')$, we have that $x_{k,0} = x'_{k,0}$. Going down the path of evaluations from $x_{k,0}$ (and $x'_{k,0}$) to $x_{0,j}$ (and $x'_{0.j}$), there has to be some i and j' along the path such that $x_{i,j'} = x'_{i,j'}$, but $(x_{i-1,2j'}, x_{i-1,2j'+1}) \neq (x'_{i-1,2j'}, x'_{i-1,2j'+1})$. This is a collision for h, which can be recovered efficiently given x and x'. Thus, if H^k is not collision-resistant, H isn't either. □

Merkle Hashing as Commitment. Collision-resistant hashing, in general, can be used directly to get a commitment scheme with computational binding (ignoring hiding for now). Given a CRHF family H with appropriate input length, the protocol between sender S and receiver R is as follows:

- To commit to message x:
  1. R samples a random hash function $h \leftarrow H$ and sends it to S
  2. S computes $y \leftarrow h(x)$ and sends it to R
- To decommit message x:
  1. S sends x to R
  2. R checks that $y = h(x)$

Exercise 2. *Prove that the above commitment scheme is computationally binding. That is, once the commit phase is complete, with all but negligible probability, there is a unique message that any efficient sender can decommit it to.*

Merkle Hashes, which is what hash families of the form H^k above are called, can be used in the same way to get a commitment scheme with the additional property that small parts of the message can be decommitted much more efficiently than decommitting the entire message. Given any CRHF family H, this protocol is as follows:

- To commit to message x:
  1. R samples a random hash function $h \leftarrow H$ and sends it to S
  2. S computes $y \leftarrow h^k(x)$ and sends it to R
- To decommit the j^{th} block of x:
  1. S finds all the $x_{i,j'}$'s in the evaluation path of h^k from $x_{0,j}$ to $x_{k.0}$, and sends to R all the labels of all the nodes in the tree of h^k that are needed to verify the hash evaluations along this path.
  2. R checks all of the hashes along the path from $x_{0,j}$ to $x_{k,0}$, and that $y = x_{k,0}$

The computational binding property continues to hold even with this local decommitment, following arguments similar to the ones we used to show that H^k is a CRHF.

Claim 2.2. *For any value of the commitment string y and any $j \in [2^k]$, any efficient sender can successfully decommit the i^{th} block to at most one value (except with negligible probability).*

Proof Sketch. Suppose there is a sender S^* that sends R some y in the commitment phase, and then, for some $j \in [2^k]$, can prevaricate in the following sense: there are distinct $x_{0,j}$ and $x'_{0,j}$ such that S^*, depending on its input, can claim that the j^{th} block of the committed value is either of these. By feeding it the right inputs, an efficient algorithm can recover these two values. Following the arguments in the proof sketch for Claim 2.1, this can then be used to find a collision for h. Thus, if the family H is collision-resistant, the probability that S^* can do this (over the randomness of choosing h from H) is negligible. □

2.2 Constructing Arguments

We can now state the theorem capturing Kilian's construction as follows.

Theorem 2.1. *Assume there exists a CRHF family $H = \left\{g : \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}\right\}$, where the description of any function in the family takes λ_H bits. Then, every language in NP has an argument system with communication complexity $O(\lambda \log |x|) + \lambda_H$, constant soundness error, and perfect completeness.*

Take any language $L \in \mathsf{NP}$ that has a PCP system with verifier V_{PCP} that makes q queries and has a small constant soundness error δ. The argument system, as hinted at earlier, is as follows given input x. Below, k is set to be such that the PCP proof is of length $2^k\lambda$, where λ is the chosen security parameter.

1. V samples a random hash function $h \leftarrow H$ and sends it to P.
2. P, who is given a witness w for x, uses it to compute the PCP proof π. It computs $y \leftarrow H^k(\pi)$ and sends it to V.
3. V runs the PCP verifier $V_{PCP}(x)$ to get a set of queries $(a_1, \ldots, a_q)$, and sends them to P.
4. P finds the blocks that contain each queried index a_i, and decommits that block to V.
5. V checks that all the decommitments are valid. It then recovers the answers $(b_{a_1}, \ldots, b_{a_q})$ to the queries from the corresponding decommitted blocks, and accepts if and only if V_{PCP} would accept given these responses to its queries.

Perfect completeness of this system follows from that of the PCP system for L. Efficiency follows from the state-of-the-art PCP constructions for NP – q is at most $O(1)$, $|\pi|$ is at most some $\mathsf{poly}(|x|)$ and can be computed in $\mathsf{poly}(|\pi|)$ time, and V_{PCP} runs in $\widetilde{O}(|x|)$ time. The running time of V is thus $\widetilde{O}(|x|)$ plus $O(k) = O(\log |x|)$ hash evaluations. The communication complexity is also easily seen to be as in the theorem statement.

Soundness is slightly more complicated to show, but not too hard either. Suppose there is a prover P^* (deterministic, without loss of generality) that manages to make V accept with probability 10δ, where δ is the soundness error of V_{PCP}. Then, for at least a δ fraction of hash functions h, when V sends h as its first message, P^* can make it accept with probability at least 9δ (Markov's inequality). But any efficient algorithm can only find collisions for a $\mathrm{negl}(\lambda)$ fraction of h's. This means that, for at least a $\delta - \mathrm{negl}(\lambda)$ fraction of h's, there is a unique answer b_a to each query a that P^* can decommit to. All these b_a's can be put together into a single candidate PCP proof π^*. This π^* will then make V_{PCP} accept with probability at least 9δ, which is a contradiction of the PCP's soundness. Thus, the argument system has soundness error at most 10δ.

Exercise 3. *Prove the soundness of the above argument system formally.*

References

[DEL+21] Irit Dinur, Shai Evra, Ron Livne, Alex Lubotzky, and Shahar Mozes. Locally Testable Codes with Constant Rate, Distance, and Locality. https://simons.berkeley.edu/events/breakthroughs-locally-testable-codes-constant-rate-distance-and-locality, 2021. [Online; accessed 11-October-2021].

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In S. Rao Kosaraju, Mike Fellows, Avi Wigderson, and John A. Ellis, editors, *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 723–732. ACM, 1992.