

[SQUEAKING]

[RUSTLING]

[CLICKING]

Yael Kalai: Today, we're going to continue. We're going to talk about a new notion, a really, really interesting notion, in my opinion, called probabilistically checkable proofs. I'll explain what it is. So we're going to define this notion, and we're going to construct it from the GKR protocol. So you'll kind of see not only the definition, but you'll be able, after this class-- the hope is that you'll be able to construct a PCP. And if you need to construct a [? visual ?] proof, you'll be able to do that.

So that's going to be the first part of today's lecture. And the second part of today's lecture is actually going to be the first time we actually use cryptography. So, so far, this class has not been at all about cryptography. Everything was information. It was actually complexity theory. But today is going to be the first time we're going to move into cryptography.

And as we'll talk, we'll see that doing this information, theoretic proofs where you say, OK, no prover can give you a false proof, that is limiting. OK, we can't do everything we want using this notion. And therefore, we're going to go into a new notion of soundness, which is what we call arguments or computational soundness.

And we're going to relax the prover and say, an all-powerful prover, an all-powerful prover, may be able to cheat, but the guarantee we give-- hey, Tony-- is that a computationally bounded prover cannot cheat. That's going to be the guarantee we get. And we'll see, once we relax the soundness requirement to a requirement that's only computational sound, then we'll be able to get a lot more, and the life will kind of open for us, and we're going to have a lot of new kind of constructions and things that we were unable to do information theoretically.

So this is the plan for today. Questions before we start? I haven't seen you guys in a while, so it's good to see you. A lot of things changed since the last time I've seen you. I'm probably going to be a bit distracted today, so I'm sorry. OK.

OK, let's start. So what are-- no questions? PL/ So let's start with PCPs. What are Probabilistically Checkable Proofs? So the idea of probabilistically checkable proofs is to take a proof, like a classical proof-- think of NP. In NP, we have an instance x and a witness. This witness is like a proof for x being in the language.

To verify this proof, you need to read the entire thing and do some computation, like this verification circuit that checks that x and w is in the relation. There's a circuit that checks this. OK, what is the idea of a probabilistically checkable proof?

The idea here, what we want, is to take this proof and convert it into a new proof and maybe larger actually than the original proof. So maybe the original proof was of size m . Let's say the instance of size n . This may be of size much bigger. Still, we want it to be polynomial in n and m .

So we just made it bigger. So what's good about it? So what's nice about this proof is, the idea is, you don't actually have to read all of it to check it. And that's weird. So the idea here-- yeah, the proof may be a little longer, polynomially longer, but you only need to read a few locations to check that it's valid.

So to verify-- often PCPs are denoted by π -- a PCP proof given x , all the verifier needs to do is query. He doesn't actually need to read these things. He queries a few locations, a few bits, checks, a few bits of the proof, and then applies a verification circuit that depends only on these bits. And importantly, the bits he chooses are random.

The prover doesn't know. Otherwise, it's like shrinking. If he knew, the prover knew, oh, the reference is going to only check in bit 17, 22, and 25, then he would just give you-- it's like you'll have a succinct witness. And, of course, we can't take arbitrary witness and just make them succinct. We don't believe that's possible. Yeah.

AUDIENCE: So it's [INAUDIBLE] only useful if m is much greater than n ?

Yael Kalai: OK. So the question was, is a PCP used if m is much greater than n ? And the answer is no. Actually, think of it, we can even use it where m the length, w is n to the epsilon-- even less than n to the epsilon, even n to the little o of n . Now, this will still be polynomial in n . So actually, you took the proof and made it actually more than polynomially larger, because it's polynomial in n and m . And if the proof here smaller is n to the little o of 1, then this is poly n . It's actually more than polynomial bigger.

So you think, what did we gain? We took a short proof, made it so much bigger. What did we gain? And what we gained is, the hope is, that you can query this, the verifier can query this, in very, very few locations, read only very, very few bits of this proof.

AUDIENCE: OK, [INAUDIBLE]. Does the verifier also have to read x then?

Yael Kalai: Yeah, the verifier--

AUDIENCE: [INAUDIBLE]?

Yael Kalai: All of it. So, yes. Good. That's a good question. So the question was, does the verifier have to read all of x ? Because I said he only needs to read a few locations in the proof, does he need to read all of x ? And a priori, of course he needs to read all of x . Because even if there's one bit in x that he doesn't read, what if that bit is-- it's not clear what the sentence is, what the statement is.

The prover needs to know, what am I reading? What are you proving to me? But there are extensions called PCP of proximity, which I'm not going to get into in this class, where actually the verifier also doesn't read all of x , but the guarantee is not that x is in the language. The guarantee is that the oracle that is given to the verifier, there is something close to it that's in the language.

So there is a notion of extension of the PCP where the verifier doesn't even need to read all of x . The guarantees there are not very easy to state, but it's a very useful notion. Actually, it's used a lot. It's called PCPs of proximity.

OK, so that's what we're going to do today. And let me just mention, before even we start the formal thing, let me just say, this is really an amazing line of work. Today, we know how to take any witness, convert it to a probabilistically checkable one that the verifier only reads 3 bits-- 3, 3 bits-- of this proof and is convinced, with [INAUDIBLE] probability. $7/7$ is the parameter. And then, if you want more, of course you repeat. But even with just 3 bits, you get some soundness, and that's really amazing.

So we're not going to see the 3-bit proof, but we're going to see a polylog n proof. So what we're going to show, the construction we're going to see today, is how to go from a witness to a probabilistically checkable one that the verifier only needs to read polylog n of these bits and verify. And to go to the constant, there's additional tricks of recursion and so on that one needs to do to get there. But we're not going to cover that part.

OK, ready to roll sleeves? OK, let's-- so let me just define the complexity class. People often think of it as a complexity class, even though, for us, because this is more of a proof system, cryptography class, we always want efficiency. We don't care just about complexity. We want to say that actually we can take any witness and officially convert it.

It's not just about existence. We want to actually use these things. And let me just jump ahead and say that these things are used. It's not just a theoretical.

Even though everything we've seen so far in the class is very just beautiful mathematically, it seems a very kind of away from practice. But actually, it's not. All these things that I've shown you so far, that you've seen so far, are things that actually are used, and the PCP as well. All this machinery are things that's use, so we actually do care about efficiency. But let me first define it as a complexity class.

So we define the complexity, the complexity class, PCP, c for completeness parameter, s for soundness parameter, r for randomness parameter, and q for query parameter. So this is a class. This is the class of all languages L that have a probabilistically checkable proof. But I'll explain exactly what I mean.

So actually, let me see, these are all the-- let me write it formally. These are all the languages L such that there exists a PPT, a Probabilistic Polynomial Time verifier-- Actually, it's a PPT oracle machine, because it takes the PCP as an oracle machine V , such that-- so here are the condition. The completeness says that for every x in the language, there exists a PCP π such that the probability that V , the Verifier, given Oracle access to this PCP π and input the instance, there exists a PCP such that the verifier will accept, it will output 1, with probability at least c . That's the completeness parameter.

For us, the PCP we'll see will have completeness parameter 1. So we want to say that a language L has such a PCP, if there is a verifier V , efficient verifier, it takes oracle access. So if x is in the language, there exists some PCP so that the verifier with oracle access to this PCP, given this input, outputs 1 with probability c . Yes.

AUDIENCE: Maybe I'm jumping the gun a bit, but what does the oracle look like? What are the inputs and outputs?

Yael Kalai: OK, good, good, good. So OK, good. No, you're not jumping, so good question.

So this π , you should think of it as a string. It has bits b_1 -- Maybe I'll call it π_1 , π_2 , up to π_i , I don't know, n . And the verifier can ask, give me π_1 , give me π_2 , give me π_3 . So it just has oracle queries. He can query this string.

That's what Oracle-- it's not really-- don't think of it as a function. It has oracle access to the string, and he can just ask for specific bits from the string. Great question. Thank you.

OK, good. And the soundness guarantee says that for every x that's not in the language-- any guesses, by the way? So you need to define the PCP here. How would you define soundness? Yeah.

AUDIENCE: I guess for all π , the probability that a verifier V with ϵ formal ϵ access ϵ to the π accepts x is at most ϵ .

Yael Kalai: Wonderful. Exactly. You want to say that for every π star-- that's often how we denote bad things, π prime π will do π too-- now there's a cheating prover. He tries to fool the verifier. He cannot. So no matter what he tries to give him, the verifier will reject him with probability or will acceptability smaller than ϵ . That's the soundness guarantee.

And ideally, we want to push the completeness to 1, the soundness to 0. Great. And then, the complexity is-- now, we want to say, in terms of complexity now will go to V uses r bits of randomness and q oracle calls. And moreover, we'll see this is important for us. This is often in the definition, though not always, but often this is part of the [INAUDIBLE], q nonadaptive.

What do I mean by nonadaptive? What I mean is the way V works, he gets input x , he then chooses kind of i_1 up to i_q locations in the PCP. OK, so [INAUDIBLE] the V . He doesn't even look at the oracle yet.

He has an input x . He first computes q locations. He then goes to the oracle, asks these q locations. And so V_1 just decides these locations. And then, you can think of V as being like V_1 and V_2 . V_1 doesn't look at his Oracle. He just looks at an input and using randomness chooses locations. And then, V_2 kind of just given x , i_1 up to i_q and π i_1 up to π i_q decides to accept or reject.

OK? So this is how you should think of the verifier. The verifier has access to the proof π . But what he does, he does-- because a more general verifier will have x , decides on location, first location to the i_1 , Look at π 1, then, hmm, interesting. OK. In that case, I want to see location i_2 . Hmm, I get π i_2 . Interesting.

Given π i_2 , I want to see location-- that would be an adaptive verifier. But often, in the PCP literature, when we define PCP, we require the verifier to be nonadaptive. And this nonadaptivity is kind of a property that we like for applications. And we'll see that today, and an example for why. When we'll construct succinct arguments, we'll use the fact that it's nonadaptive.

But let me just say, most PCPs, all PCP I know, are nonadaptive. Maybe there are other. Was that your question? Yeah. So there may exist. I didn't do a full literature search. But the PCPs that are the most common ones that we talk about are all nonadaptive. Yeah.

AUDIENCE: The V 's know what capital N is.

Yael Kalai: Yeah. Yeah, yeah, yeah. So V , yes. Yeah. So very good question. Let me repeat for the video.

So the question was, does V know what capital N is? So the answer is yes. And V knows the language. So he knows what small n is. Well, he got small n . He knows small n . The language has there is m So V gets small n . So he knows what n is.

The language is associated with a witness size m of n . So he knows that, because he knows the n , so he knows the witness size. And the PCP itself is associated with N , which is, you can think of it this way. So the answer is yes.

These are kind of fixed. These m and n are fixed parameters that are defined by the PCPs, and the prover knows them. The verifier, the prover, they all-- yeah. Yes.

AUDIENCE: Do we gain or lose something with V adaptive.

Yael Kalai: Yes. Good, good, good. So the question is, do we lose something? And the answer is, we do. OK, so what do we lose and what do we gain is your question. What we lose is for application. So when we'll go to-- in the second part of today's class, we're going to construct succinct arguments, and we're going to construct them from PCPs. And you'll see that if our PCP is adaptive, this becomes a problem.

So we do gain. Now, you're saying, what do we lose? So we do lose. Sorry, we do lose if we make it adaptive. Now, we're saying, but maybe we gain something. Maybe we construct PCPs that are much better if you make them adaptive.

And I don't think so, but I'm not sure. Maybe one can actually argue that you don't gain anything, but I haven't thought about it. But it's not something that-- making it nonadaptive, for all we know, is not a price. It's just our PCPs? Happened to be that way. Yeah.

AUDIENCE: Is there a version where you choose [? the ?] [? instance ?] [INAUDIBLE]

Yael Kalai: Good, good, good. Very, very good question. So I think maybe I misunderstood, but maybe your question was, is it clear that these have to depend on X ? Can they even not depend on X . And the answer is yes. Actually, in the PCPs, we'll see, they don't depend on X . The PCPs we know, actually, these, they only depend on n and the instance size or begin.

But they don't actually depend on the instance. And the PCP I'll construct today, you'll see that. So we'll get back to that. Really, really great questions. Yeah.

AUDIENCE: High level, how does that work? Because you think that if the prover knows what indices you're going to query in advance, you could construct [INAUDIBLE].

Yael Kalai: Of course, of course. Great. Wow, guys, you're really lifting me up. This is great. Thank you.

Yeah, 100%. So what you're saying is, of course, if the prover knew the locations you're querying, he will cheat. The thing is, these locations are random. The prover doesn't know what they are. They just don't depend on x .

AUDIENCE: Oh, OK.

Yael Kalai: But they come from a distribution. It's very important. If these were deterministic, there's no way. If you can do a PCP where these are deterministic, it's like you took NP and you just made it n times much smaller. Like, we don't believe that can happen. So no, there's no way we believe you can construct PCPs for which the query distribution is deterministic. It's just fixed. This has to come from a distribution.

But the question is, does this distribution depend on x , or does it only depend on the parameter n ? And in what we'll see, it actually only depends on n . It doesn't depend on x . So that's what we'll see in a minute.

Great question, guys. Any more questions before we-- yeah.

AUDIENCE: So [INAUDIBLE] in V_1 and V_2 , or do we really only care about the [INAUDIBLE]?

Yael Kalai: OK, good. So actually, usually V_1 is randomized and V_2 is deterministic. So you can think of it that way, that V_1 is randomized, V_2 is deterministic. OK. Great.

So now, let me tell you what we know and what we're going to construct. So here's what we're going to prove today. Here is a theorem that we're going to prove, that PCP with a, let's say, completeness $1 - 1/2$. I'll put some as $1/2$. We can get smaller, any constant smaller, but I'll just put $1/2$, because by repetition it doesn't really matter-- you can get a PCP where the query complexity is-- so, OK, we can get a PCP with $\text{poly log } n$ query complexity and $\text{polylog } n$ randomness complexity and query complexity for all of NP.

That's what we're going to see today. So we're going to see how for any NP language, we can construct a PCP where the verifier, the amount of randomness he uses, is $\text{polylog } n$ bits of randomness and $\text{polylog } n$ queries. And then, we get completeness 1 and soundness $1/2$. Yeah, Tina.

AUDIENCE: So [INAUDIBLE] this is not the usual [INAUDIBLE]. Usually, you want proof-- you want the randomness [? complexity to be ?] [? log ?] [INAUDIBLE]

Yael Kalai: Right, right, right. Yeah, yeah, yeah. So you're right. So often-- OK, you're right. The way I wrote it here-- so OK, why do people care about randomness complexity one can say. Well, who cares about the random complexity? And usually, the way I define PCP, and that's why I said as a complexity theory, this is how it's defined, if the verifier has only $\log n$ bits or order $\log n$ bits of randomness, then it means that the PCP is only $\text{poly } n$ size. Because, look, go for all the $\log n$ bits of random, there's only n of them. So that's as much as you can query.

So that means the PCP is upon it. and that's what we care about. OK. For me, I'm going to care about-- actually, for me, I'm going to add a requirement. Actually, I didn't write it here, because it's usually not defined. But I can add a requirement that says-- OK, I'll add it here tiny. This is a requirement that we add in cryptography.

Often, what I'm writing now is not added in the definitions. But let me write another cryptography, another that given for every x in L and witness-- if it has a witness. w . OK, so suppose this is a language in n time, I don't know NP or N time T -- so it has some witness-- I want it to be the case that, given x and w , you can efficiently compute π . So not only there exists π of $\text{poly } n$ size, you can actually efficiently compute it.

And because I require efficient computing, I actually don't care anymore so much about the randomness, the number of randomness bits. But yeah, that's a very good point. Thank you. Guys, thank you for the question. This is great.

OK, so that's what I'm going to prove. Even though, let me just mention, what is known is you can make this order $\log n$, and you can make this 3 . So it's really remarkable. If you think about it, wow, you can take any proof.

So you know how, I don't know, take Fermat's last theorem. Take the hardest proof that we in mathematics. One can write them in a probabilistically checkable way. They can just take 3 bits and verify. And if you're not happy with probability $1/2$, repeat it as many times as you want. You want to be sure that he can cheat with probability only $1/2^2$ to the 500 ? Repeat it 500 times.

And each one is like a very simple check. It depends on 3 bits. So it's really remarkable.

AUDIENCE: Never mind, sorry. [? N choose 3 ?] probability is way too big, but I guess it doesn't really matter to have all the possible--

Yael Kalai: Oh, you're saying it's too big for theory, for practice. Yeah, but we're still in theory land. Yeah, in practice, these PCP, you want them to be linear. That's the ideal if they're quasilinear. Even if they're quadratic, people don't like them. You really want linear, quasilinear PCPs for practice. But for here, poly is fantastic. We're not we're-- let us stay in the cloud a little bit. OK.

So OK, let's construct this. Ready? OK. So we have the 3, but we're going to construct for the poly log poly log. And actually, here's what's amazing. You actually already know how to construct the PCP. You just don't know that.

And how do you construct a PCP? It's really only the GKR protocol. That's it. So you can unfold GKR, and I'm going to show you exactly how to make it a PCP. So to do that, but before I now recall GKR in a second on that board, to do that, let's first-- so I want to say any language has a PCP-- let's focus on 3SAT.

OK, so 3SAT is is an NP-complete language. Let's just construct a PCP for 3SAT. Sets. And what's nice about 3SAT is that-- I mean, any NP language you can convert to 3SAT, so it's nice in any NP language-- you have a circuit, a verification circuit-- so a circuit verifying, you can think of it, given w verifying that it's a valid witness-- of very small depth. OK, polylog n depth.

And this is important. Because if you remember the GKR protocol, which we're going to use to construct this PCP, kind of the communication complexity grows with the depth. So we want the depth of the circuit to be small. But any NP language you can convert into a 3SAT, and 3SAT has-- actually, the depth of 3SAT is like actually only 2. But you need to do, little or's and then one big and.

But if you remember, in GKR we wanted fended in 2. So once you have fended in 2, it's like order log. OK, so this is depth. You can think of it as being log n or I'm OK with poly log n . It doesn't matter. The point is, it's small.

And the reason why I write also polylog is because it's actually log. However, this circuit is also-- and this I'm not going to get into, but it's logspace uniform. And now, you can consider the circuit. Change it a little bit the way Rachel taught you two weeks ago. If you don't remember, it's OK. You can take it as a black box.

You can assume that the circuit, by kind looking at a universal version of it, that it has the gates add i and mult i are efficiently computable. And that's what we'll need. So if you don't remember things from two weeks ago, it's OK. I'll remind you what add i and mult i is in a second. But you take any 3SAT, you can convert it into 3-CNF. So now, given a witness, you just check that it verifies.

Hi. Hello. Hey. Wow, all my kids coming in. OK. So we're talking about how to construct PCPs from GKR right now. So what we're proving is that-- we're going to show how you can take any NP language-- in particular, we're focusing on three 3SAT-- and we're going to show how you can convert 3SAT into a PCP where the query complexity is polylog n . So converting to PCP, the verifier needs to query only polylog n locations.

We know you can go all the way down to 3, but today we're going to solve polylog. This is the randomness complexity. This is the query complexity. We're focusing on completeness 1, soundness $1/2$. OK.

So we're going to now construct a PCP. And before we construct, the PCP essentially uses GKR. So let me first, on this board, remind you quickly what GKR looks like, because we're going to use it to construct the PCP. OK. Cameraman, it's OK to move?

CREW: Yeah, no problem.

Yael Kalai: OK. Good. So let's just recall the GKR. How does it work? We have a circuit. There's an input w . We're going to think of w here as the satisfying assignment. OK? Remembering, the GKR, we assume the verifier has w in his hand, the input in his hand. Here, he doesn't. w is going to be the witness for the satisfying assignment. But that's OK.

Now, I'm going to construct the PCP proof. OK? I have my witness, my satisfying assignment. I'm going to construct a probabilistic check. I'm going to convert w into a PCP, into a probabilistically checkable proof. OK, here's how I do it. I look at the circuit that checks that w -- actually, if you want to think of it as a completely new for me you can think of having x and w . So x and w , and it checks that kind of ϕ -- sorry-- x , which is ϕ , and it checks that ϕ of w is 1.

OK? This is logspace uniform circuit. OK, now what do I do? So here's me. I know w . We both know the 3SAT formula. And we assume that the circuit has addition gates and multiplication gates.

What I do, for every layer, I compute the values of the layer. OK? So I compute for every layer. Remember, I have V_i . I assume every layer, let's say, the same size without loss of generality. We set size s . We called it s . So each layer has s wires.

And we compute all these wires. For every layer i , we think of these wires as going from H to m to $0, 1$. So H is some set and, we assume that H to the m is s . OK? So we just encode these s wires in a m -dimensional cube of size H , and we just write all the values of the wires. OK?

And then, what we do, we look at the extension V tilde which extends to a field F to the m to F . And this is kind of the low-degree extension. So we take any F that contains the set H , and we extend via the low-degree extension. So these are things we covered.

Now is the first time-- you remember, when we talked GKR, I told you, OK, why not do $0, 1$ to the m ? 0 to the $\log n$ it's much natural to think of binary representation of s as opposed to this H to the m which is what H . And I told you, oh, often we like parameters. We like to think of H as $\log n$ -- sorry, $\log n$, and m is $\log n$ over $\log \log n$.

And if you'll see, H to the m just becomes n . Because you can think of H to the m is $\log n$ to the n , but $\log n$ is just to $\log \log n$ to the n And the $\log \log n$ cancels out, so you have 2 to the $\log n$. So you have-- oh I did everything as s . Sorry. S . The n here is s . S . S . S .

So you have s . Now, remember, I told you, oh, we'll need these set of parameters as opposed to the $0, 1$. It's natural to think of H as $0, 1$, and m is $\log n$. That's much more natural. We're used, as computer scientists, to think in binary. Now, we're thinking of $\log s$. It's like, really, how annoying can you be?

And the reason is it's for this. Now, we need it. Why? Now, we take f that is going to be the extension of size $\log s$. OK? And we'll see the poly will need to be big enough. We'll talk about it in a second.

But the field, we don't want more than $\text{poly } \log s$. And the reason is we want F to the m to be $\text{poly } s$. So we know H to the m is s . If F is only poly bigger, than we have this is $\text{poly } s$. And that's important for us.

OK, note, if H was just 0 1, so the size of H was 2, we couldn't keep F poly and we couldn't keep F constant. You'll see, F will need to grow with m . And so, for soundness, $m F$ will need to be bigger than m times H . Actually, there's also d that will come in. It's significantly bigger.

So if we take H to be 2, we'll need to take F to be at least $\log n$. And then, we'll have F to the m is $\log n$ to the power of $\log n$. That's super poly. And I want F to the m to be poly. OK. So we do this. So now, what is the PCP?

Actually, I'll right there the PCP. But now, let's just finish recalling the GKR. So what does the verifier do in the GKR, the prover do in the GKR? He computes all these low-degree extensions. And then, the GKR consists of d phases going from the output layer to the input layer, where in each phase we run two sumcheck protocols in parallel where the verifier uses the same randomness. And what the sumcheck protocol does is it reduces from checking two values here to reducing checking two values in the layer below.

And we go kind of from checking and until we get to the input, and the input the verifier can check on his own. That's the idea. OK? OK. Now, let's just recall, in a sumcheck protocol here, each sumcheck protocol, if you remember-- maybe you don't, the details are not so important right now-- you go from V_i in the extension, which is, let's say, V_i tilde of some z , and you write it as this is kind of sum of points in H to the m , because that's the definition of extension, times some $\chi_P z$ -- this is the function that checks equality. Again, it's not important.

But what's important to me to show is that-- and then you do sum over $P w_1 w_2$ in H of m , and you convert V_i to either-- if this is a plus gate, you convert it to the plus of the wires below with some duplication. You check the multiplication. So, in other words, what this is, or I should say what this is-- let me just open this. What is $V_i P$?

It's if it's an add $P w_1 w_2$ -- so if point P is this just the addition of the two wires below, then you replace P with w_1 with V_i minus 1 w_1 plus V_i minus 1 w_2 . And plus, if it was not an add gate, if it's a mult gate, then you replace it with the multiplication of V_i minus 1 w_1 times V_i minus 1 w_2 . So the idea was-- and all this is times this χ_i .

But again, how do we go from layer i to one layer below? We take that element in the extension, we think of it as a sum-- that's the definition of a low-degree extension. It's the sum of the elements in the circuit itself with some function, with some weight. And now, we look at the value. What is the value of this gate?

Well, if it's an and gate, it's the sum of the two children. And if it's a mult gate, it should be the multiplication of the two children. And so add i just outputs 1 of these-- if w_1, w_2 are the children of P , 0 otherwise, similarly mult. But we do everything in the extension. We look at everything in the extension, because we want to think of it as a polynomial.

And once we have these are all polynomials, then this is just a sumcheck. And we saw sumcheck how to do it efficiently. So now, the point is GKR consists of only these $2d$ sumchecks. So now, let me go to my PCP.

I'll erase this. So here is our PCP. Essentially, my PCP is just GKR but opened up. Completely, completely unraveled. OK, so here's my PCP. I have my x and w . Here's what I do.

I'm going to compute all the V_i tilde, the extension, the values of all the gates in each wire, for every z -- for every z in F to the m . It's important, all this is going to be part of the PCP. So if F to the m is more than polynomial, my PCP is superpolynomial. so F to the m has to be polynomial. This is already size polynomial. Yeah.

AUDIENCE: Do we still have [INAUDIBLE]

Yael Kalai: Again, sorry?

Audience: Do we still have the restriction on the [INAUDIBLE] [?] to have that [INAUDIBLE]

Yael Kalai: Oh, yeah, yeah, yeah. Yeah, yeah, yeah OK, good, good. Yes. So for GKR, we had a restriction that the circuit-- to do GKR, we said GKR is like proving that bounded depth circuit [INAUDIBLE] is y . And we had a restriction that the circuit has to be bounded space uniform. Or, in other words, we needed to ensure that add_i and mult_i , this extended n_i and extended mult_i , are computed efficiently.

3SAT has that property. Again, I didn't go into the details, but the 3SAT is very uniform. Given a ϕ and x , it's very uniform to check. It's a logspace uniform. And then, we can convert it, as Rachel did two weeks ago, to one that by increasing the depth by poly factors, so keeping it poly log, you can make add_i and mult_i be efficiently computable. So yeah, great. That's a very good question. So yes.

Good. So this is going to be part of the PCP. But that's not all. By the way, we should also give-- we give this for every i from the input layer 0, 1 up to d . This is output. And this is the input layer. We have to give him the input layer, because the poor verifier doesn't have w . So he doesn't have actually the input.

Before, in GKR, he had the input. Here, he doesn't have the input. But we're going to give him the low-degree extension of this w . OK, good. OK, so I'm the verifier, what do I do? I need to verify the sumchecks?

Who do I interact with sumchecks? I'll put all the possible answers in the oracle. So now, we do $2d$ sumchecks. So for every possible, for every i and for every z in the GKR protocol, we may need to do a sumcheck. Because in the sumcheck protocol, in the GKR, how does it look like?

The verifier first decides, OK, prove the output layer is correct. Let's say the output, the z is fixed. Fine. Then, he chooses randomness, randomness, randomness sum of sumcheck. It reduces the checking z -- d minus 1 in a random z . The z is determined by the random point of the verifier. So which z do I put? And I want to show a sumcheck for every possible z .

So for every possible z , I'm going to put in the PCP a sumcheck. So I'm going to give a sumcheck. I'm going to add all the possible transcripts for a sumcheck corresponding to $V_i z$ or maybe to V_i equals V_i tilde z . So for every i and z , I'm going to give a sumcheck with respect to randomness r from the verifier. So for every i , for every z , and for any randomness r of the verifier.

Now, how much randomness does the verifier use in a sumcheck protocol? So remember, the sumcheck is sum over H to the $3m$ kind of variables. OK? It's sum over P w_1, w_2 , all of them in H to the m . So the r is going to be for every r in F to the $3m$. Because in the sumcheck protocol, what you do, is every single H , every single kind of-- you have $3m$ variables. Every one you give a random field element to reduce-- get rid of one of the sums.

So the verifier will choose r_1 up to r_{3m} . So $3m$ field elements. So for every r , I'm going to actually give you the transcript. So now, I know I'm going to give you the entire transcript.

But here is the thing, I'm going to have a sumcheck for all of them. And now, the verifier. OK. But we need to be careful. We don't just give for every r a sumcheck. If I knew r , I would cheat.

So what I mean here is for every sumcheck, for every V_i , for r_1 , I'm going to give all the outputs for r_1 , all the kind of answers of the sumcheck for r_1 . Then, for each one of them, for every r_2 , all the possible answers. So no repetition. You can't have two different elements.

You can't have one value if you had r_1 , r_2 , but later different r 's, or r_1 or r_2 -- for every possible partial transcript, you give a value. That's what the PCP has in it. So for every-- I should say, yeah-- i , for every z , and for every, maybe I should write it as, r_1 up to r_{3m} , I have a transcript. I should have all partial transcripts of sumcheck with respect to V_i equals V_z and randomness r_1 up to r_j for every j between, I don't know, 1 and $3m$.

OK, so for every r_1 I will say, what would be my answer? So now I have F to the m answers for every one of these answers. For every r_2 , I'm going to tell you, what is the answer? For every-- sorry, I said it wrong.

For every r_1 , I would give it all the answer. I have F . Each one is one field. So for every r_1 , I'll tell you the answer. That's F options because there are F options for r_1 . And then, out of these F -- for each of these F 's, for every r_2 , I'm going to give you what the prover will say. That that's F squared.

And then, for each of these, I'm going to give you r_3 . So really, what this PCP-- for every V_i -- for every such V_i -- or for every kind of V_i z , for every possible r_1 , I'm going to give you-- this is over r_1 . I'm going to give you the possible answers in the sumcheck protocol. If the prover in the-- if the verifier in the sumcheck protocol, the first randomness was this r_1 , this r_1 , this r_1 , I'm going to write all of these in the PCP.

And then, for each r_1 , every possible r_2 , I'm going to write my answers. And then, for each r_1 or r_2 , for every possible r_3 , I'm going to write my answers. It seems like an exponential tree, and it is an exponential tree. But the depth is so small. So this is a huge tree, but this tree is only of size F to the m -- or to the 3-- I'm sorry-- which is polynomial. It's S to the 3rd.

And you're right that this is not good enough for practice, but then there is a lot of optimization in the GKR that make it quasi-linear and, today, even linear. So you can actually optimize this to get linear time, but-- or-- but-- or quasi-linear as a PCP. But for now, we don't care about these polynomial is good enough for us.

So for every i and for every z , we give this tree. That's the PCP. That's it. OK, so-- yeah?

AUDIENCE: Doesn't the sumcheck protocol rely on the fact that F is big?

Yael Kalai: Good, good, good, good, good. Fantastic. You're like, wait, that's a problem. If I require F to be small, do we even get soundness? That's a great question. So the answer is, barely, but yes.

[LAUGHTER]

So let's see. What's the-- OK, what is the soundness that we get? For it to be sound, Each sumcheck, so that the-- OK, so let's talk about soundness. Maybe I'll put it here. Let's talk about soundness here for a second because you raised a very-- but you know what, actually, before we talk about soundness-- I'll jump to soundness.

You have a very good point. We'll talk about soundness. But just to make sure the construction is clear, so let me just reiterate one last time I'm a PCP prover. I want to convince you guys that I have a satisfying assignment to a three-step formula. I have my w . What I do, I construct a low-degree extension of my w , my witness.

But also, I have this entire circuit that verifies this witness. And I'm going to construct a low-degree extension of each and every layer in this circuit. Now, the way I wrote it, actually, I will construct a low-degree extension of x and w . That's kind of-- if I think of the input is x and w and to construct a low-degree extension of both. But you-- the verifier knows x . So if I construct a low-degree extension of w , he himself can construct a low-degree extension of x and w together.

So in the PCP, because I don't want to-- in the PCP, I don't want to put x in there. The verifier has x . And if I have the freedom to put x in there, and you don't read it, then I can put the wrong x . So I'm just going to put V_0 is just going to be of the witness. I'm going to think of x as fixed.

So I'm just going to construct a low-degree extension of w . And then I'm going to construct a low-degree extension of each and every layer of the circuit. And I'm going to go-- for every layer and for every z , I'm going to construct all the possible prover messages in the sumcheck protocol.

Now, because the randomness, the options for the number of possible random bits from the verifier is only F to the $3m$ because there are $3m$ variables, and f is the field. So it's only after the $3m$. Then, this tree is only going to be of size f to the $3m$, so it's not too bad. And I put all of them. I put all the possible sumchecks in there.

So it's F to the $3m$ for every-- I have this size for every i and for every z . OK, so for every i , i is only D . It's this is small, and z is another f to the m I pay. Fine. So after for. So for every z and for every i , i have the sumcheck.

Now, what the-- what is-- so that's the PCP. But now what's-- what does the verifier do? What is V ? I need to define-- I told you how to construct this π efficiently. And it's-- but what does V do?

So what does-- what V does, he says, I'm going to pretend that I'm interacting with a GKR prover. So he says, look, what is the output? He's going to read the output, VD . And he's going to expect a 1. If he sees a 0, so that's one query. I'm going to read the output. If it's 1, good. I'm happy.

Now he's going to go from verifying D verifying D minus 1, D minus 2, D minus 3. Now he wants to talk to a prover. He said, OK, here is my randomness, my first message of the sumcheck. Give me back something. OK. This is the prover. The prover is fixed in the PCP. OK, so he gets back an answer, da, da, da, da, da, da. Good.

Now, here, these at the end, this defines V_i minus 1 Z . This should be all kind of elements in V minus 1 z . Now, actually, here, what you have here is, at the end of the GKR protocol, you're left with two V 's, right? So if you verify here, you'll have-- at the end, you'll have V , I don't know, 0 and V -- z_0 and z_1 . You'll verify-- you need to verify two points because we did two sumcheck all the way.

So fine, he sees these two. He checks. There's these. He checks that they're equal. This is only 1. So we're giving him all the low-degree extension of V_i minus 1. He checks, and oh, yeah, this is-- corresponds. This corresponds.

This is just to ensure that he doesn't put different ones in different-- here, he may-- in this branch, he may put the value of Z_0 to be something, and in this branch, he may put the value of Z_0 to be something else. That's not good for us. So we just check because he put aside all the low-degree extensions. So he checks that it's equal to what he gave us. And we continue with that. OK?

One second. Actually, I'm not sure. Maybe what I said is incorrect. I'm not actually sure that this is a problem. Even if he gives us different-- let me think about it for a second. Actually, I'm not even sure that we need-- we can just put the sumchecks. Even if he gives us-- even if he gives us-- no, OK, yeah.

I want to say, even if it gives us, in this branch and this branch, different things, why do we care? It's like, we ran the sumcheck, and the sumcheck is sound. But we do care a little bit because then, in the next round, which-- let me think. You know what, I'm not actually-- what?

AUDIENCE: [? My ?] [? understanding ?] [? is ?] that you sort of just lay out the whole randomness of the verifier, right? And then verifier [INAUDIBLE] actually [? interacting ?] [? with the ?] prover just like [INAUDIBLE].

Yael Kalai: Exactly. You let all the-- so yeah. So I don't think, actually, you need to give these specific. You can just give-- for every V_i tilde, you give-- but it's the same because you give for every V_i tilde. So actually, it's-- I think these two are the same. So-- OK.

What I-- I guess what I said is, these can be-- maybe V_i minus 1 of z_0 appears here and appears here, and they're different. Are we worried? We're not really worried, because it's like-- you think of it, it interacts with the prover. The prover-- the cheating prover may give him different z here and different z here.

But later-- the point is, later, you'll have-- in V_i minus 1, you'll just continue. So actually, you just need to give-- but really, I think the two are equivalent, so it doesn't really matter. OK, so this is between how you verify. And at the end, at the end, at the end of the GKR, the verifier needs to read the input. He needs to check the input in some locations. He doesn't have the input.

But he does, because we gave him the low-degree extension of w . So once he has the low-degree extension of w , he can compute anything. So the verifier needs to compute a low-degree extension of x and w , but any point in the low-degree extension of x and w is a point in the low-degree-- is a sum of points in the low-degree extension of x and point in the low-degree extension of w .

So every point in the low-degree extension of x and w , you can write as a weighted sum of a point in the low-degree extension of x and a point in the low-degree extension of w . So once he needs to read the point in the extension of w , he has it here. So that's how the verifier does.

So he really-- he's-- it's interesting, this PCP, he-- as if he's talking to the prover, but the prover is sitting in the sky in the PCP. The prover's strategy has just kind of opened up in the oracle. OK? OK, great. So now, any questions before we talk about soundness. Yeah.

AUDIENCE: [INAUDIBLE] this protocol, the buffer versus the [INAUDIBLE] graphed. And then [INAUDIBLE].

Yael Kalai: So right. OK. Good, good, good question. The verify-- great question. The verifier needs to check also that all these things are actually low-degree extension.

So what if I said here, oh, you put V_i tilde z ? How do I know that you actually put a low-degree extension? Maybe you put a very high degree polynomial there. Maybe you just put an arbitrary truth table of size after them. That would be problematic. So you're totally right that the verifier-- so let me write what the verifier does maybe before the soundness. Let me actually write it.

So the verifier, he runs. The first thing he does, he runs V of GKR with its oracle. So he runs just the GKR, but he thinks of the oracle part as kind of the prover he's talking with. And the other thing he needs to do is he needs to check that for every i , V_i tilde is of low degree-- is--

AUDIENCE: Does the original verify of GKR do that? Again, doesn't it sound as if GKR just relies on [INAUDIBLE]?

Yael Kalai: Yeah. What you're saying is that if-- you're right. So what you're saying is-- what you're saying is if he gave V_i tilde, that was high degree. Then you're saying the GKR would fail. You would be rejected by the GKR.

AUDIENCE: It should be. Otherwise, it just doesn't work.

Yael Kalai: It should be rejected by the GKR. I think you may be right, but actually, I'm not 100% sure now.

AUDIENCE: [INAUDIBLE] that E_i is of high degree, but still agreed with the higher or more degree.

Yael Kalai: Yeah. So here's the concern. Here's the concern. The concern is we want to make sure that the verifier is behaving like-- we want to make sure that this V_i tilde is of low degree. Why? Sorry. 1 second. Yeah, actually, I-- I don't think we need to check that V_i tilde is a low degree, because-- I agree that we need to check that V_i tilde is of low degree, but I think GKR checks that the tilde is a low degree. So I actually don't want to recheck it.

So all I want to do here, I think GKR tests it for you. So we're convinced that the GKR is sound. Let's not recall why it's sound. But let's remember-- we learned that it's sound. So GKR is good. Now, what are we doing? Essentially, what we're saying is, verifier, I'm giving you all the possible answers.

What is the prover? The prover is the tree. You can think-- what is a cheating prover? A cheating prover, for every possible question, he gives you an answer. For every question, he gives you an answer. That's what it does. So now you're saying, look, I'm going to give you the tree. Now, in general, this tree is really, really big. And even in GKR, if you give the entire-- if you just open-- GKR has D rounds, D rounds of sum check. If you just open the sum check, it will be terrible.

However, GKR has this property that it's kind of memoryless. What do I mean by memoryless? You can compute the first sum check. Then you don't care what happened before. All you care is what is V_i minus one that you cut, and you check those. So now I'm saying-- so you know, actually, let's think about this. First of all, we can say, open GKR to a huge tree.

Any prover-- this is true in general. If you had any interactive proof, you can think of a prover, right? All the possible answers. Think of its strategy as a huge, huge tree. Now, of course, you get soundness. You interact with the prover. What's the difference?

You can do that with GKR. But of course, if you do this huge tree, it's going to be gigantic because GKR consists of D kind of subprotocols. Each one is a subject. So now if you think of like a D , [INAUDIBLE] then you'll be too big. But I'm saying we don't need to, because GKR has a property that, actually, it consists of this subprotocol. And then all you need to remember-- you don't need to remember which branch you're on here. All you need to remember is what are the values here. That's all you need to remember.

So there's no point in repeating. It's like you're repeating the tree for no reason. So all you need-- there's a way to open this prover in a more efficient way. So by not giving the entire tree, giving kind of a more efficient kind of opening, where all the V_i 's-- all the V_i minus $1z$, I'm going to say, OK, forget about it. Let's just go from V_i minus $1z$ and continue. Why do you need to repeat this kind of tree 100 times? But you can go from here to simulating an entire GKR prover. Yeah.

AUDIENCE: So if I understand correctly, you're saying, actually, the tree of GKR is such that most of the parts are copied. And so you don't need to repeat the copies. You can just look it up.

Yael Kalai: Exactly. Exactly. Exactly. Exactly. Actually, in some sense, we're restricting the prover here more than he's restricted in GKR. Exactly. Because in GKR, he can be, you know what? I'm going to remember the R 's, if he's cheating. And based on these R 's, I'm going to give you a different strategy. Whatever. Here, we're not even letting him do that.

We're saying, look, you tell me what you are here. From here, now you forget R 's. From V_i minus 1 , what's your strategy? And now you forget. From here, what is your strategy? So we're really just doing GKR. Now, you write that the V_i minus 1 tilde does need to be of low degree. But the GKR test is for you. So you don't need to do it on your own.

But you're right. It's confusing a little bit, because if you-- if anyone is familiar with the PCP literature, there's always a low degree test in it. So it's kind of interesting this is hidden within the GKR protocol.

AUDIENCE: This kind of transformation is not only for GKR. As long as there is an efficient way to change the prover into a sequence that the verifier knows where to compare it will give you a [INAUDIBLE].

Yael Kalai: Exactly, exactly. So yeah, this proof holds. It's not specific to GKR. You can take any succinct protocol and interactive protocol and lay it open. However, let me just mention we have very few-- actually, kind of two examples of succinct protocols. So it sounds like you can take any of your choice. Yeah, any, but we don't have much choice. Coming up with these succinct, interactive proofs is a super interesting question. I'll talk a little bit more about this after we do the analysis of the PCP. So great. Any questions before we move to soundness.

So let's talk about soundness. And the question that was raised, which was a great question, is, wait, f needs to be big to get soundness. So let's really understand what the size of f needs to be. So to get soundness, we do $2d$ sum check protocols. Each sum check protocol needs to be pretty sound, because it's enough that you cheat on one of them and we're dead.

Now, each sum check protocol, this is just recall that each sum check protocol has soundness, which is H/F times m the number of variables. But for us, it's $3m$, because we have $3m$ variables. This is just the sum check protocol.

Now, for us, so you can cheat, but with this probability. Now for us, we have $2d$. D is polylog n , like the depth of the circuit. So for us, our soundness is-- I'm just doing it with union bound. So to cheat, there exists-- needs to be at least a kind of on sum check protocol for which you cheated on. So you get by union bound. So we have $2d$ sum checks, and then, therefore, the soundness is $2d$ times $3m H$ divided by F .

So now if we want soundness half-- let's say we want F so that you can repeat to increase the soundness-- all we need is that F -- so if we want this to be smaller than half, then we need F to be bigger than $12d \log n$. But that's good. This is $\log n$.

I don't know if you guys can see. $\log s$ -- sorry. This is smaller than $\log s$. This is some $\text{polylog } s$. So just make it bigger than all of them. It's still polylog . And $\text{polylog } s$ is good enough, because then after them is $\text{poly } s$. So this was soundness.

AUDIENCE: Sorry. Why did we need-- I think I just [INAUDIBLE]. Why did we need F to the m to be $\text{poly } s$ here?

Yael Kalai: Oh, oh! Good. Why do we need-- you're saying, why can't it be bigger?

AUDIENCE: Yeah.

Yael Kalai: Because-- good. Because we want our PCP to be-- so the question is, just to recap, why do we need-- why do we insist that F is not bigger? Why does it have to be polynomially related to F ? And the reason is we want the PCP to be of type $\text{poly } s$. We want the PCP to be-- we want it to be efficient. We want to efficiently convert a witness to a PCP. And the PCP grows with F to the m much more. It's like F to the $3m$. And it needs to be repeated.

And F to the m is poly -- if this is bigger than poly , if F is more than $\text{poly } H$, then F to the m will be more than $\text{poly } H$ to the m . So that's why we want it to be-- it's for the sake of efficiency for the PCP to be small or polynomial related to the circuit.

AUDIENCE: Just so I understand, so here you're saying that the randomness complexity is $\text{polylog } n$. Now, normally, you'd be like, the proof size is just 2 to the randomness complexity. But here, you've been more smart, right? You've found a way to make the proof shorter than it would be otherwise.

Yael Kalai: OK. Good, good, good. So now let's go to the-- we didn't do the complexity yet, but yeah, you're a bit ahead of me, but you're totally right. So far we've talked about soundness. Completeness is one, because the GKR is complete. It has completeness 1. So the sum check has completeness 1. So we're good. We talked about sum checks. Any question about soundness? Any question about the soundness?

Let's talk about the complexity. So the randomness complexity in the communi-- and the query complexity. So how much query and randomness do we need? So the randomness complexity is-- well, there's $2d$ -- so let's put here randomness. So there's $2d$ sum checks. So the verifier kind of interacts with his oracle on $2d$ sum checks.

So for each $2d$ sum check, he needs to toss coins. Now, how much coins does he toss? Well, he needs to generate r random in F to the $3m$. That's each sum check. You need-- so it's F -- it's like $3m \log F$ bits of randomness. Now, this is really $\text{polylog } F$. It's like $\text{polylog } s$. It's not $\log s$. This is really poly , $\text{polylog } s$, not just $\log s$. So one can be weighed.

We need to be careful, because, are we sure we're getting an efficient PCP here? It's a lot of randomness. Are we sure there's no blow up in PCP? And the reason there's no blow up, the reason we have large randomness, yet the PCP is small is exactly because GKR has this property of forgetting. The fact we actually don't need to open the entire tree, because of that, we use an additional property. The point is if you know nothing about the PCP, then, of course, as I said, if it's order $\log s$ bits of randomness, it's always $F \text{ poly } s$.

But the fact that there are more bits of randomness does not necessarily force our PCP to be bigger. We're just not going to open it trivially. If we did open it trivially, then we would look at all the random. It'd be a huge tree, like the depth. But because we do pack it efficiently, we're able to get a small PCP, even though we have a lot of randomness.

AUDIENCE: Is it because you're saying each tree is sized poly and then poly trees?

Yael Kalai: Exactly. Each tree is size poly, and they're poly trees. And we're not-- there's a more straightforward way to do it is for each tree, append, append. No, we're kind of doing them in chunks. And so we get the efficiency.

Let's go for the query complexity. I'll just put it here. So let's see. So for-- again, what's the query? He does GKR. So he does $2d$ sum checks. For each sum check, he needs to read what the answer is. Each answer, so there's $3m$ rounds that he needs to read. In each answer, he gets kind of a univariate polynomial of small degree. Each answer is not just one field element. It's actually a few field elements.

So he gets $\log F$ elements times the degree, which will be in each sum check like poly H at most. And this, again, goes back to the golden eye of what's exactly the degree of the add i multi. It's polynomial. So I'm not-- it's polynomial in H . It's polylog H . I don't want to-- this is the degree of each of the sum check polynomial. And the degree is affected by the add i multi, which we assumed the degree is poly H . But it's still, again, polylog.

So this gives-- so before we take a break, let me-- where are we now? So we studied the-- we started, actually, this class by looking at sum check. And so I started this class with a regular look that sum check is a miracle. Given sum check, it gives you everything, everything. And I hope that now you see why. So we start by sum check. And we looked at sum checks like, who cares? This sum, modal it seems like why would we need to sum over so many multivariate polynomial? Where does that come up? I don't know. It was like, why am I learning this?

So now what did we do with it? We started with sum check. We saw just by doing sum check repeatedly, you can verify any low depth computation. That's pretty amazing. And then we saw that, actually, you can recover the IP equals space space theorem, that GKR actually gives you the IP equals space space theorem. And now we see GKR actually gives you PCP. And what is GKR? It's just sum checks.

So just these sum checks, you kind of unroll them-- unravel them, and you get a PCP with polylog n queries. And not-- it's like efficient. You can take your input, your witness, and efficiently generate this PCP. And in a very-- like, a straightforward way in a sense. You do your computation. You have your witness. You do the computation. You do all the low-degree extensions. And you convince round by round. You just lay it out, and that's it. And you have a PCP. Any questions about?

AUDIENCE: So the fact that the GKR tree can be extended, that relies on just the fact that it consists of these kind of independent subjects, or also the fact that subject itself has-- like, each query is just a random--

Yael Kalai: Good, good, good. So the question is, again, let's understand why do we get this PCP that's efficient. Why do we get only poly-sized PCP? And the reason we get-- it's two-- both, twofold. You said both reasons. Reason number one, when we do each sum check, after that, we can forget about this randomness. And for each V_i consider. So we really need to write down only kind of d times F to the m sum checks. That's the depth of the circuit times F to the sum checks. That's number one.

But number two, why is the sum check? Can we write out all the sum checks? I mean, the sum check is also an interactive protocol. Can we actually open it up and write the entire thing? And the answer is, well, yeah, but at an exponential cost. But an exponential cost in m , the number of variables. But here, the number of variables is \log . That's important.

If m was polylog, we would have been doomed. But m -- or I mean, OK, it's not just \log . We set the parameters so that it's OK. That's what we needed. That's exactly-- so why can we open up this entire sum check? It's because we set F to be poly H , and H to the m is S . So we set it up so that opening the sum check requires F to the m kind of answers, F to the m answers. Each one is kind of a low-degree polynomial, but that's fine, but F to the m of them, or F to the $3m$ here, because we have-- the number of variables is $3m$.

And we just need to make sure that F to the $3m$ is polynomial. And to ensure that, we did do some parameter engineering. That's why we didn't go the natural route of taking H to be 2 and m to be $\log s$. That would be natural, but then we would not get. So we actually did need to do some kind of engineering and parameters, like an annoying fiddling of parameters.

This setting of parameters that you see over and over in any PCP construction, this is-- we're not the first. We're actually the last, if anything, to come-- to use this. This was used in the '90s in the early PCP constructions. And it's when you look at it first, like, really? Like, $\log n$ over $\log \log n$? Are you kidding me? Like, couldn't you find something nicer? But this is why, because we want these parameters to be poly. Yeah.

AUDIENCE: So since you're essentially asking for this to be like a non-adaptive prover, can you just use the same randomness for each round and then get rid of the d factor [INAUDIBLE]?

Yael Kalai: Sorry. So you're saying-- sorry. Your question is-- you're saying-- are you talking about the fact that this verifier is non-adaptive?

AUDIENCE: Or that the prover is non-adaptive, I guess.

Yael Kalai: The prover in that-- yes, like it's non-adaptive in the previous sum checks.

AUDIENCE: [INAUDIBLE] previous.

Yael Kalai: Previous sum checks. yeah, so then what?

AUDIENCE: Can you use the same randomness for each round? Like, what would happen if you use the same randomness?

Yael Kalai: No. OK, OK. That's a good question. So you're asking-- you're saying, look, we're doing the sum check d times. Can't we use the same randomness? And the answer is we cannot. Why? Because you should think if we use the same randomness for different rounds, then you should think of, I'm a prover. I'm like, oh, you're going to use the same randomness for all your rounds. I know that, because you're telling me already, that you're going to use the same randomness.

So now in the first round, I'm going to behave, honestly, just to learn your randomness. Give you-- I'm sorry. I'm not going to behave. I'm going to cheat in the first round. So of course, I'm going to cheat. The output is 1, even though it's 0. I'm going to manage to convince you in the first kind of-- in the first sum check I do, because I'm just going to lie consistently with myself.

I'll write to V_i minus-- to the next layer, and the wrong-- still inconsistent. But now I know the randomness, because you're going to reuse. You see, the point is the following. The verifier here should behave like an honest verifier in the GKR. If the verifier here, you cannot-- when you use this verifier, he essentially behaves like a GKR verifier. So now you're telling-- you're saying, look, why doesn't he use the same randomness for the different subprotocols? Because if he did a GKR, then he would fail. Then a cheating prover can take advantage of him.

AUDIENCE: Isn't there the difference here that this prover is not adaptive? It doesn't remember the randomness. All it depends on is V_i minus 1.

Yael Kalai: Right.

AUDIENCE: So you can't just be like, oh, this is the randomness. [INAUDIBLE]

Yael Kalai: Oh, I see. You're saying-- I see. I see. OK, OK. Now I understand your question. You're saying, this guy, this prover is stuck. Sorry. Now I understand what you're saying. You're saying, look, you're tying the hand of this prover. The prover in GKR, he has power. He sees your randomness. And based on that, he can continue. The honest prover, of course, won't do that. But the cheating prover may.

Here, you're saying, I'm not going to let the prover do that to me. I'm going to force him to forget. Because I forced him to forget, can I think of using the same randomness? That's a good question, actually. A great-- very good question. It took me time to understand what you're asking.

AUDIENCE: Can you use the same--

Yael Kalai: It's a great-- great point.

AUDIENCE: --that same malicious verifier you just said to break that?

Yael Kalai: Well, no, because-- so your point is the following. You know what? Use the same randomness in all the subprotocols. Now, in the GKR, it's a big problem, because the prover, he knows what you saw. And he'll use that. He knows the randomness. But in the PCP, the PCP GKR prover, his hands are tied.

AUDIENCE: Yeah. But the fact that he-- if the prover knows that you're going to do this, then he might be able to engineer all of his responses in a way that takes advantage of the correlation of the randomness.

AUDIENCE: Exactly. You think the same verifier you said. Pretend you have that verifier that cheats [INAUDIBLE] use the same randomness. Now, this tree follows that verifier's strategy. Then shouldn't that break it?

AUDIENCE: Even if it doesn't follow just a GKR soundness.

AUDIENCE: Yeah. I mean, it is-- it could be some, but yeah, there's this correlation between the randomness, which the prover would know about in advance, right? So even though he can't remember the randomness, he can engineer a strategy to take advantage of the fact that the randomness repeats.

Yael Kalai: Let me say the following to close the discussion. A, the discussion is super interesting. The question you raised is super interesting. You know, we have one P set in this class.

[LAUGHTER]

You gave me a great idea for a question. By the way, before we break, I just want to make a quick announcement. So I wrote in the website that the P set will come later during this class, because I thought it would be good to cover more material. But I know that I wrote it pretty close to the break. And maybe that's-- so maybe I'll put it out earlier, give you two weeks, just because I don't want you to sit with it too long. It's just annoying. But if you feel like the time just doesn't work, and you want an extension, just let me know. I don't really care, to tell you the truth. I just want you to think about it and have fun with the material. Yeah.

AUDIENCE: When we argue soundness, are we using any sort of union bound lexeme?

Yael Kalai: Yeah, exactly. We're doing a union bound. Yes, exactly. Because in the soundness, we say the following. We say that in the-- so essentially, we're talking with the prover and the GKR prover. And we're saying for the GKR-- even like a tied GKR prover, as you mentioned, it's a GKR prover with tied hands.

So for this prover to cheat, he needs-- so the output is false. That's kind of-- he cheated. So somewhere, he needs to cheat in one of the GKRs. One of the sum checks, he needs to go from false to true, somewhere. Because at the end, you're verifying. You're verifying. He put here a witness. So he put in this guy a witness. That witness is false, because there's no satisfying assignment. So he put something that's actually false.

Oh! 1 second. There is something. OK, sorry. Now I remember. There is something. There is a low-degree test to do. We'll get back to that. Sorry. We'll get back to the low-degree test. But so you're talking with a GKR prover. And this prover, the only way he can convince you is to go from a false statement to a true statement. Because at the end, you verif-- and to do that, you need to break one of the sum checks.

So to break one of the sum checks, you have $2d$ sum checks. And I'm using a union bound to say that the probability that you managed to break one of them is 2 to the d times the probability that you managed to break one of them.

AUDIENCE: But then back to the discussion where you're sharing randomness, so does union bound help here?

Yael Kalai: You're saying-- oh! Because of the--

AUDIENCE: [INAUDIBLE] dependence of them.

Yael Kalai: OK. So is this related to this question?

AUDIENCE: [INAUDIBLE]

Yael Kalai: Actually, I don't think so, because union bound, in general, it can be correlated. A union bound in general says that the probability that this or this, I don't care how they're correlated. It's small or equal the probability of this plus the probability of this. So the correlation here, the union bound is very easily used in any situation. You just have the probability that you have an event. You cheated on test 1. You cheated on test 2. You cheated on test 3. I don't care how they're correlated. The probability that you cheated on one of them is smaller or equal then.

But actually, before we break-- sorry-- there is a low-degree extension.

AUDIENCE: [INAUDIBLE]

Yael Kalai: What?

AUDIENCE: In the first [INAUDIBLE].

Yael Kalai: Exactly!

AUDIENCE: Because in GKR, you can just use brute force.

Yael Kalai: Good. OK, OK, OK. Sorry. Good. I love you guys. So yeah, V_0 is low degree, because in GKR, the verifier has the input, which is the witness. He computes a little extension by himself. Here, the prover is giving it to him. What if you didn't give a low degree? So we do need one low-degree test in the input. Other than that, it's GKR, black box. But we do need this.

And by the way, note that we're using the GKR, the GKR has actually properties. And I want to go back to the succinctness, because, actually, you're saying, can't you do it with any succinct protocol? And the answer is maybe. But there's also this point, which is problematic, that in GKR, it's nice because you just check one point in the low-degree extension. And we can verify that you do the low-degree extension. There is a low-degree test that I didn't talk to you about, but we'll mention that after the break.

So there's a GKR. You can verify. The verifier, to verify, all he needs to do is take his input and compute only two points in the low-degree extension of that input. This is actually used here, because now the prover-- the verifier doesn't have the input, the witness. The prover has it. He's going to put in the sky. And now you can't read the entire input. That's a witness. But you can read what the kind of succinct protocol tells you to read. And so this part is important in the GKR.

Questions before we break? So after the-- we'll take a 5-minute break. And then we'll talk quickly about the low-degree test, wrap up the information theory, and go to cryptography.