[SQUEAKING]

[RUSTLING]

[CLICKING]

YAEL KALAI: Before we go to the test, I just want to point out this PCP is nonadaptive. What are the queries of the verifier? He behaves like a verifier of the GKR or, in other words, a verifier for sum-check. And sum-check is a public coin protocol. So the messages don't depend on anything. You just choose some randomness, and that defines the queries.

So it is a nonadaptive-- the questions you ask don't depend on the answers you get. This is because the sum-check is public coin so that we do get nonadaptive. And as we said, though, we do need to check that the witness sitting in the PCP is of low degree. So let me just do that quickly.

So how do we check it? So the prover, what he does, there's the H to the m. He should put some witness here, 0, 1, 0, 2-- w0, w1, w2, and so on. So all the up to wm-- m is up to-- let me call the number of bits in the witness s. We'll just have a lot of 0's appendage just so-- because m mistaken to be log s, so not to confuse.

And now he's giving kind of he takes the extension. So this is f to the m. And this should be-- this is just what it should be. This is what the prover does. This is the low-degree extension of H to the m, which should be of degree H minus 1 in each coordinate. And I want to check that.

Now here's the sad truth. I can't actually check this. This thing is low degree with degree H minus 1 in each coordinate. I can't check it because maybe it is, but here, he put something bad. I can't check it. I mean, I'm not good. The verifier has small queries. So he can't actually check that this is low degree. That's impossible.

So what does he do? What does he do instead? He's going to check that it's close to low degree. So here's the guarantee. The guarantee we want is to say that I'm going to check. I'm going to do a test. And if the test passes the guarantee I want, not that this is low degree, but it's close. There's another polynomial of low degree that agrees with this on most points. So we have a low degree test is a check.

At the end of the check, what the verifier-- the guarantee the verifier has is the guarantee we want is that with high probability, there exists a low degree polynomial, W tilde such that-- W tilde such that for most z or the probability of a random z that Wz equals is greater than 1 minus epsilon for some epsilon. That's what we want to check for most.

Now if we got this check, if we succeeded in this, if we succeeded in this check, then now what we know is that-- let's think in our head that this is-- he put this up there. Yeah, he put this. But they're so close. Let's pretend he put this up there. Anyway, the probability that we'll ask and a z for which they're different is only epsilon. Mostly, they agree. So all we need is to verify that there's-- because we check V0 and a random z at the end, most likely, it's the z's for which they agree.

So all we need to check is that they agree almost everywhere. The fact that there are few locations that they don't agree-- A, there's nothing I can do about it because if I don't query them, I don't know if they exist. I don't know what's sitting there. But, actually, I don't care that much because they're not going to-- I'm not going to query them anyway with high probability.

So really, all I want to say is that there exists a w tilde that is low degree and agrees with high probability. How do I do that? How do I check that this W, that this V-- completely malicious V0 tilde-- behaves close at least-- as I said, we can't assume it's totally, but close to low degree-- agrees with the low degree polynomial in many places. Yeah?

STUDENT: [INAUDIBLE]

YAEL KALAI: Exactly. Exactly. Let's check on some random. Let's do a random sample. The way it's done-- so there are many low degree tests out there, by the way. But one kind of common one is the following. I'm going to take a random line in f to the m, and I'm going to check-- I'm going to ask, open this random line. Now this random line should be of degree m times H minus 1. That's what it should be. I'm going to check that. That's the degree. If it's not the degree I reject, then you're immediately rejected.

So now what I know is that, on a random line, you agree with the low degree polynomial because, otherwise, you're rejected. Now one needs to argue that because you agree on a random line with a low degree polynomial, why does that mean that you're actually close to a low degree polynomial all around? But this is the test. This is really one of the low degree tests that's out there.

So this is sitting in the PCP. The verifier will choose a random line, and he will test-- you can even test m times H, random points on this line. He'll take m times-- or you can test this plus 1. It doesn't really matter, depending on the soundness you want. But you can even take this plus 1. This defines polynomial of low degree of this degree. And now you check that the additional point agrees with it.

Or, if you want, you can ask the entire f is only anyway, only polylog s bit. So you can just read the entire thing and see that, by interpolation, that it's a low degree polynomial, and if it's not, you reject.

And then one needs to check, though. So a random line agrees with a low degree polynomial. Why does it imply that everything agrees with the low degree polynomial? That's a leap-- or with high probability. So I have a good homework problem around that. So let's leave that alone. I have a lot of this-- I have a lot of good homework problems today, which is nice because it summarizes all the information theory part of the class. Any questions before we proceed? Yeah?

STUDENT: So is there some argument here that if the polynomial is low degree, then it corresponds to some weakness in this, like, PCP?

YAEL KALAI: So if the argument-- if you can argue that this is close to low degree, then take the actual low degree that it's close to. Now this defines a witness.

STUDENT: Oh, by evaluating on each-- I see.

YAEL KALAI: Yeah, and now this is going to be the witness that you violated on, exactly. Exactly. Great. Yeah?

STUDENT: [INAUDIBLE] we don't really need to do a [INAUDIBLE] circuit because if the [INAUDIBLE], and you can just do a single subtract to do it.

YAEL KALAI: To do the AND, you're saying?

STUDENT: Yes, some kind of things right there because we don't need to subtract the first [INAUDIBLE].

YAEL KALAI: Yeah, so what you're saying is-- what you're saying, look, there may be ways to do optimizations here. The way I showed you is very kind of generic. It said, well, this is low depth. Now I say wait. It's not just low depth. Look, it's a very specific circuit. It's one big AND with three ORs, and maybe you can do it much better. So let me say you're 100% correct. There's a lot of optimizations on GKR. I kind of showed you just kind of a feasibility result.

But, as I said, these things are used, and they're not used as I showed it because they're optimized. So there's a lot of optimizations that they do, and a lot on the circuit level, as you say. What's the best way to arithmetize the circuit? No, we don't want to just maybe open it to just random-- degree 2, degree 2, degree 2. Maybe we don't want to do exactly the go doc. Maybe we want to do something else. How to generate the circuit to make it the most efficient possible-- there's a lot of optimizations on that part.

So yeah, I showed you. And, as I said today, you can use GKR to be like-- the GKR protocol can be linear time prover. They manage to get to-- actually, I don't actually-- I know how they got to quasi linear. The linear is actually a new work that I didn't yet have time to dive into. But, apparently, they have really nice optimizations that they use. And most of them, I think, is exactly-- and the circuit level around ideas that you mentioned. Great. So any other questions? You guys have great great questions, so this is fantastic.

So I want to say a few things that I didn't talk about. Maybe they'll show up in the homework. Maybe not. And it's related to questions that people asked, that Leo asked and others. So, first of all, I said-- I showed poly log. I said we can go all the way down to 3. Let me just mention, 3 is the best we can do. We cannot go below 3. From poly log n, you can do kind of log log n.

Using recursion, you can do PCP on the poly log n queries. You can recurse. And then, from the log log n to 3, you can do-- there are these exponential-sized proofs. But to take a log log n and make it-- there's exponential-sized PCP with only three queries. So you apply an exponential-sized PCP to this log log n sized PCP. They have all these cool tricks, and they manage to get to 3. Three bits. Three bits. Can you go to two bits? No. Yeah?

STUDENT: Is that only for soundness half, like if--

YAEL KALAI: No. Any constant soundness. No--

STUDENT: What about for negligible soundness?

YAEL KALAI: No, constant soundness. Constant soundness. For constant soundness, we can get 3. The constant, I think, is better than half, but whatever. But you can get 3. And can you get constant soundness for 2? No. No, I think you just can't. So I don't want to get-- yeah, I don't want to get-- and can you get 1? Definitely not. So can you get 2? No.

However, you can get 2 if you make each the answer to each query like a block. So if you want two queries where each query gives you an answer bit, you cannot have PCP. And the reason actually-- intuitively, the reason you cannot have a PCP is because two SAT is easy. If you have only two kind of queries, intuitively, the reason why it's because two SAT is easy. I don't want to go into. Maybe it can be homework, so let's leave that.

But what you can have-- and we do have-- it's not just do-- we do have PCPs that have only two queries, but the answers are large. I mean, OK, you can have one query if the answer is the size of the witness. OK, that's easy. But we can have two queries where the size of each answer is poly log. OK, but not one. Or constant for all I know.

So maybe, we'll do some homework kind of to get intuitions around these things. Yeah.

AUDIENCE: Actually, I think we do have to be qualitative if we're allowed to have imperfect completeness. [INAUDIBLE]

YAEL KALAI: Oh, if you allow to have imperfect completeness, then you're right. Sorry. Everything I'm saying is for perfect completeness. Once you allow to reduce completeness to 1 minus epsilon, things change. So what do we know, yeah, with two-- if you have. Yeah I'm not I don't remember off the top of my head what's two. But what's the soundness. Two queries, constant soundness. 2 bits.

AUDIENCE: One third, two third.

YAEL KALAI: One third, two third. OK. OK.

AUDIENCE: [INAUDIBLE]

YAEL KALAI: Yeah, yeah, yeah, yeah. OK, yeah. You're right. So here, I'm assuming-- yeah. You're right. Yeah, good point. Thank you. Yeah, so if you're willing to relax completeness, we can get 2. But perfect completeness, you can't. Yeah.

AUDIENCE: Wait. When you say, "blocks," and when there are two blocks is possible, are you talking about contiguous bits, blocks [INAUDIBLE].

YAEL KALAI: Yeah. So yeah. Think about it-- each query you read-- so the PCP consists of instead of bit, bit, bit. Think about it like a word, a word, a word, a word. And each word is poly log and bits for example. And now, you can read only two words. So you can say I want to read word i and word j.

AUDIENCE: OK.

YAEL KALAI: Yeah. Yes.

AUDIENCE: Don't we not have perfect completeness right now because of the low-degree test that has [INAUDIBLE]?

YAEL KALAI: OK, good. No. So good question. So the question is, now, do we have perfect completeness? And the answer is yes, because if you're honest-- perfect completeness means if you're honest, you'll be accepted with probability 1. So if you're honest, you are going to give w tilde, which is the low-degree extension of your witness. And when I take a random line, it's going to be of low degree. For sure. Everything is low degree. If you're malicious, then--

AUDIENCE: So the soundness is [INAUDIBLE].

YAEL KALAI: The soundness. Exactly. You lose-- you're right. Good point. What you mentioned is you say, because we do the low degree. We lose a little bit in the soundness because maybe you're a little malicious and we don't catch you. Or and that goes, yeah. So you need to make the soundness a little bigger so that you can eat up what you lose a little bit in the low degree. Yeah, yeah. Great question. Yeah.

AUDIENCE: When you're doing the protocol to bring the standards down, can you just use the same tools and just check the different randomness?

YAEL KALAI: Yeah, yeah, yeah. Good. So the question is when you repeat, so let's say you're not happy. We have soundness half. We're not happy. What do you do? You need to actually repeat the entire PCP or-- no you don't because the prover will just give you the same PCP over and over. So no. You keep the same PCP. All you do is you run the verifier, the PCP verifier, fresh with fresh randomness more times.

So if you have soundness happen, you want to have soundness 1 over 2 to the k, what you do is you run the PCP verifier k times. So it generates one set of queries, you run them fresh again, another set of queries, fresh again. Each time is going to be rejected, probably half, independently of the other time. So you just get that is rejected. He's accepted probability 1 over 2 to the k. Yeah. So just the verifiers needs to repeat. The prover is not involved in this. Great question, guys. Any other questions?

AUDIENCE: What's the reference for this two-query thing?

YAEL KALAI: I can put it in--

AUDIENCE: [INAUDIBLE]

YAEL KALAI: Yeah, I can put it in Canvas, in the website. OK, if I don't remember, remind me. OK. So before we move to crypto, I just want to wrap up kind of the information theory part and say that in some sense, I taught you almost like, OK, so what's missing?

What did you not get a chance to see? You didn't get a chance to see the three query. So you didn't get a chance to see the recursion, and the Hadamard, and this nice way to getting three PCPs. There's another beautiful, beautiful result by Reingold, and Rothblum, and Rothblum from 2018. '16? '18? Yeah.

AUDIENCE: [INAUDIBLE] '16, but now when you look at it, it says '18 because they-- the [INAUDIBLE].

YAEL KALAI: Yeah, OK. Well, it started in '16. And since then, there's work. But essentially, here's the question. This is kind of a question that's still missing. And here's the question.

The question is, we have efficient, doubly-efficient protocol for bounded depth. Can we construct doubly-efficient protocol for bounded space? Now, look. We know that we cannot hope to have doubly efficient, or not doubly efficient for that matter, for more than p space. We know that ip equals p space. So you can't have interactive proof be the prover efficient or not efficient for more than p space.

OK, so we're stuck with p space. Now, you can exchange. You can go from depth to space. Actually, there's a way to convert depth to space and vice versa. But it doesn't preserve efficiency. So you can go from bounded depth to bounded space. Now, you must go because I showed you a double-efficient double rooted bounded space if it required more than poly space, something in the world would go wrong because we know ip equals p space.

So the point is we do know. We can convert a bounded-depth computation to a bounded-space computation. But this conversion doesn't respect the efficiency of the circuit. So if you take a bounded-depth circuit and convert it to a bounded-space circuit, the bounded-space circuit, or bounded-space computation, I should say the bounded-space computation now will just be very, very long.

Even if the bounded-depth circuit, let's say, was only size s polynomial, the bounded-space computation can be exponential. OK, so it doesn't respect kind of the efficiency of the computation when you do the transition from bounded depth to bounded space, or vice versa. So now, there's a question. Can you do-- let's say you have a bounded-space computation, not a bounded depth. Yeah, you can convert a bounded depth and pay a price. But you don't want to.

Can you do a bounded-space computation, make it doubly efficient? OK, so here is kind of a goal. And those who've been working with me for a while maybe traumatized by this goal because I tormented them. I think this is one of the most interesting problems out there. It's really frustrating to me that there's no solution. I'll give you an incentive in a second. But here's the-- OK. So here's my question.

If you have any time T space S computation, t of n, s of n. Here's the goal. Goal, we want an interactive proof such that prover runtime is efficient, poly t. We want verifier runtime, or-- let's count, OK, verify runtime. Well, the poor guy has to read the input length. So x plus times I'll be nice with you, times whatever. It can be even poly. I'm happy even with poly if you want to put poly here. I'll be generous-- times poly and the space, not depending on T.

So he can read the input and run times the space. And usually, we want the complex, the communication complexity, to grow only like poly space. But if you want to put x in there, fine. Be my guest. This is what we want. This is open. So we want a doubly-efficient interactive proof for p space.

Let me tell you what Reingold and the Rothblum brothers did. They almost did this. But here's this annoying part. The verifier runtime and the communication complexity is S, great, but times T to the epsilon. And this epsilon has to be a constant because the communication complexity grows with 2 to the something, o tilde, I think 1 over epsilon, or something like that. So there's exponential growth in epsilon. So you can't take-- you can maybe take epsilon, like, 1 over log, log, or log, log and like log, log, log n over log-- 1 over log, log n.

Think of it as constant. You don't really gain by trying some constant. It's not giving you much. So they present the result as epsilon being a constant. So this is really annoying because now the verifier, if T is polynomial, is super polynomial, the verifier is not even efficient. So this is really for p, actually. I want to get rid of this and do for big T. OK, this is still an open question. OK, now, I need now OK. I'm begging you guys, a plea for the smart, all you smart MIT students, or Harvard, or at BU, or Northeastern, or whoever you are. This is a personal favor.

Can someone please solve this? And I know I need to give you guys incentive. So beyond probably I don't know, like a ACM PhD award or whatever, that's nothing. But I'm willing to give you a $500 prize.

AUDIENCE: Whoa.

YAEL KALAI: So--

AUDIENCE: You're outbidding [INAUDIBLE].

YAEL KALAI: That didn't give a prize on this, right? He gave like a fake cheap prize or something.

AUDIENCE: But it was, like, $15.

YAEL KALAI: OK.

AUDIENCE: He was a good [INAUDIBLE].

YAEL KALAI: OK, so this. You get rid of the T, do this. You can take pictures. So you can actually work on this, please. OK. Let's move on. OK, so this essentially concludes the unit on information theory. OK, there's another proof system that I didn't talk about, which is a multi-prover interactive proofs. The reason I didn't talk about them is because they're really equivalent to probabilistically-checkable proofs. They're not much different.

Maybe I'll put something along these lines in the homework. I don't know. I need to think about it. But that's another thing we did not cover. But the really, an equivalent thing to PCP. But it's another proof model that was considered in the literature. OK, so that concludes information theory. So that's all we know. We're stuck with p space. Actually, we don't even-- and we can get a very efficient verification. But then, we have a very long proof like the PCP.

You can verify things super efficiently, like fantastic. But you need to write somewhere this gigantic PCP proof. That's the PCP. And note, it's very important that the verifier, that this PCP is written. If the PCP is not written somewhere, and he's talking with the prover and telling him, look, I know you have a PCP written there somewhere, can you give me locations i, j and k, a cheating prover will choose the locations based on i, j, and k

And then for i, j, k, he'll put in location i 0, and maybe for i and j prime k prime, he will then put location i 1, and cheat. So it's very important that the PCP is written. And now when you ask query i, it's what's written there. The answer cannot depend on query j or query k. So it seems like it's a very nice model. It's fantastic. But where is this huge thing going to be written?

OK, so if we want-- at the end of the day, here's our goal really. What we're after really is to take an actual proof, like a witness, and shrink it. It's big. I don't like big. Shrink. Make it small. Here. Now, it's smaller. Now, of course, we can't do it. So or can do it. So now we start changing the model and say, OK, you know what? Let's try interactive. You know what? Let's try PCP. All these are very nice. But they all have really strong problems.

OK, like the interactive proof one, it's very, very specific. It's an interactive proof. I can talk to Tina. I can talk to Sue. I can talk to Leo. But it's per person. What if I want to give a proof to the world? I can't do an interactive proof. I want to prove some of my proof. OK, now I can do PCP. I can post that, but it's really big. So if I want to make it succinct, I can't.

So I want to be able, again, my goal, to take a proof and just shrink it and put it somewhere. Now, of course, you're like, OK, it's nice. It's good that you have desires. Well, it's good that you want. But actually, what's really interesting is that using the magic of cryptography, we can get these things.

So now, what I'm going to do next is we're going to dive into the world of cryptography and see how we can use cryptography to get towards kind of these succinct, non-interactive proofs.

OK, so let's start. So the first way we're going to change the proof model is to say to consider what we call computationally-sound proofs, also known in the literature as arguments. So let me explain. So instead of talking about statistical proofs, which is what we talked about so far, we're going to think about computationally-sound proofs. OK, so what is a computationally-sound proof? It's a proof that has a much weaker guarantee, soundness guarantees, than the standard one.

So the completeness is the same. It's a proof you want to prove. If something is true, you succeed. But that soundness proof, so this is also called an argument.

So the difference between a actual proof and an argument is in soundness. That's the only difference, the soundness condition. So let me just tell you what the soundness condition is here. So it's a computational soundness condition. And what it says is that, so let's say for some language L, what it says is that for any computationally-bounded, only if the prover is bounded.

If you know that the prover is bound, resources are bounded, its computationally-bounded runtime. So only for a prover that's runtime is bounded, and for every x not in the language, the probability that p star, and v, and x that the verifier accepts is small. OK, let's say negligible. OK, smaller than half. Whatever. Usually in the language of arguments, we convert this constant to negligible and non-negligible. We'll talk about that in a minute.

But now, we say, you know what? How do we get around all these negative results? So the idea, we say, you know what? Actually, why do we care that there exists a time 2 to the 1,000 prover that can cheat? Nobody can run at that time. OK, there's not enough molecules in the universe. Like, there's no-- so these things, they don't exist. So why do we care? Let's only consider provers that run in bounded time. Now, you can say, what is bounded? We'll talk about that.

Is bounded, what? There's an input length n? Is n and n cubed? 2 to the n? What's the bound? So yeah, each-- we'll talk how we bound it. But the first idea, let's bound his runtime. Let's consider a prover that is runtime we know is bounded. Now, let's try to argue soundness only against such provers. OK, so you take the bound to be something that you believe is realistic. Yes.

AUDIENCE: Do we have some distribution over x? Or literally--

YAEL KALAI: No. For any x. So yeah. So the question is, just distribution? No. I still want to say, look, I don't want to say that you can cheat on any x. So look, I want the same guarantee. I want to have the same soundness guarantee. Say, look, nobody can convince you of any false statement.

AUDIENCE: But couldn't your prover just, like, memorize the proof of some easy x?

YAEL KALAI: OK. Good. Good. Good. Right. So you're saying, wait. What do you mean? If you have an x, you take a proof, and you put it there. I'm defining here-- yeah, so OK. So what you're saying is, you can't have this non-interactive. You're saying, if you want a non-interactive proof, this is impossible, even computationally. Why? I'm a cheating prover. I'm going to have one bad x, x on the language, I'm going to hardwire one bad proof because we assume it exists. It's just hard to find.

Fine. It's one. So I can kind of think of a non-uniform model. I'm going to hardwire it, and now I'm going to give it. You're right. We do not have non-interactive proofs, computationally-sound proofs. But we almost do. Almost. So we'll talk about what I mean by almost. Yes.

AUDIENCE: Is that one random [INAUDIBLE]?

YAEL KALAI: Yeah, exactly. We, OK. So I'll jump ahead and say, the way we have it is we assume some common random string, some common reference. Let's say we all agree there is some hash function. And we're going to use this hash function when we construct the proofs. And now, we're going to argue the only way you can cheat is by breaking this function. So we are going to at the end get exactly what we want. So this is kind of jumping ahead. We will actually be able to construct these succinct proofs and non-interactive.

However, we assume that both the verifier and the prover kind of agree on some common reference string that both agree. It's universal. It doesn't depend on x. It's some hash function. And we all agree it's same as chapter 56, whatever any kind of hard for those who do cryptography and are familiar with these kind of off-the-shelf hash functions. For those who don't, don't worry. Whatever. We agree on some reference string.

And now, we're going to argue that if a cheating prover can come up with x not in the language, and a false proof, it means he broke something about the common reference string. So this kind of-- the idea is why-- so I already said it actually. But when you first think about it, you want to say, look. I want to get around the fact that ip is only p space, all these kind of barriers.

I want to be able to construct these succinct proofs. Now, we know we can't. So we need to change the model. And one way we say no, let's consider bounded provers. Why not? Let's assume my prover runs in time less than 2 to the 500. Would anybody object? Is there any proof of the runtime more than 2 to the 500? No. So fine. We didn't lose anything. The question is, what did we gain? Is it easier? OK, we said the prover has to run time most 2 to the 500. So what-- how do we use that?

And the reason we use that, the way we use that is using cryptography. Now, we can say our prover cannot break a cryptographic assumption. So we add some cryptographic assumption to our proofs. And we can assume that the prover can break it. Now, whoo. Like, a whole new world opened up to us, and which we can use. OK?

So what I'm going to show you next is actually how to use-- so here is a theorem. I'm first going to show an interactive protocol. Still interactive, but very strong. So here's what I'm going to show you. This is a theorem due to a Kilian and Micali 92 and 94, which are really the kind of results that kind of laid the foundation for this field.

And what they showed is the following. They showed how to take any PCP and some little cryptographic tool called the collision-resistant hash function, I'll define what that is, and take these two and construct a succinct four-message, so not non-interactive, four-message argument. So computationally-sound proof. So essentially, what they show is, give me this long PCP that can be efficiently checked. I'm going to use cryptography to kind of shrink this PCP, put it in a little box, send it to the verifier, and then convince the verifier that's what in this box is an accepting proof.

So they use cryptography together with PCP to construct with the following, assuming collision-resistant hash function, I'll define in a minute, a cryptographic primitive. For any language in NP, there exists a four-message argument system. OK, now let me define the argument system more formally, because here was just kind of a bit-- And now, I'm-- OK. So when we talk about-- once we use cryptography, when cryptography comes along, then we use some cryptographic primitives.

And these cryptographic primitives have some security parameter associated with them. The security parameter intuitively tells you how much-- so when I say I want, let's say, 100 bits of security, we usually in cryptography, essentially what we mean as a prover that runs in time 2 to the 100 cannot break the scheme. Or in time depends on the security, cannot break the scheme.

So whereas statistically, we say when we say, we want a, I don't know, security epsilon, we say probability of breaking, any-- you can have whatever power you want. But the probability is epsilon. Once we're in crypto land, usually the security talks about the runtime. Like, what is the bound you place on the prover's runtime?

And usually, we think of the soundness as negligible. OK, so negligible meaning smaller than any polynomial. I'll explain in more detail. But so when we talk about a cryptographic primitive, it's associated with the security perimeter. And we don't-- in practice, when they write things, they write, usually they just set security perimeter to be 128 bits, 256 bits. They set a number.

But here, we're in complexity land. So we have a parameter, lambda. That's the security parameter. And we keep it as a parameter. And in real world, the people set it to their heart's desire. But we're going to think of it as a parameter here. OK, so here's the guarantee. OK, so this four-message argument system, let me start with the complexity, actually. The proof of runtime will be poly N, the instance length, and lambda security parameter.

And v, runtime, is going to be only polling security parameter. And also in our case, also in the plus, VPCP runtime. So of course, he needs to read the input, and he needs behave like a PCP verifier. But the PCP verifier is very efficient. The problem is the Oracle. So now, there's Oracle. But his runtime is like the runtime of the PCP verifier, so which uses some PCP.

And you can think of the PCP I just showed you, poly log n, poly log n. So we use some PCP. So that's the complexity. It's four messages. The prover runtime is efficient. And poly of course and in it. And the verifier is very efficient.

AUDIENCE: So [INAUDIBLE] given the PCP or something?

YAEL KALAI: Oh, Oh, Oh, sorry. P. Yeah, OK. sorry P given x and w and the witness. Yeah, thank, you. The prover is given x and the witness in the language. And he's going to be efficient. He's going to run in time n and lambda. Let me just mention, OK I mentioned that in a second actually. So the completeness is 2

It just says that for every X, and L, and witness. W, the probability that P with witness W and V, both of them have X. And security parameter, there are some security parameters involved. The probability that the verifier outputs 1 is 1. So the verifier always accepts if you're honest. If the prover is honest, he has a witness corresponding to x with any security parameter he accepts. OK, actually, almost, it's assuming-- no. It accepts. No ifs and buts.

OK, what about the soundness? Soundness says that for any p star of size, now, you can say, for instance, if size poly lambda, poly in the security parameter, then you can say, let's say that for every x not in the language, the probability that p star-- now, he doesn't have a witness because it's not in the language. This is a cheater. I want to say for any p star that tries to cheat. But he's bounded. He's poly lambda.

And for every x not in the language, the probability and x in lambda that the verifier outputs 1 is very small. It's negligible in lambda. And negligible in lambda is just a way to say it vanishes faster than 1 over poly for any poly. OK, this just means that for any poly p, there exists some lambda star such that for any lambda bigger than lambda star, p lambda, the negligible, is smaller than 1 over p lambda.

So it vanishes faster than any poly if you go above some constant. OK? So the neg 0 it vanishes quicker than 1 over poly for any poly. That's the definition of negligible. Yeah.

AUDIENCE: Interested in for what size means. So we're assuming all these things are circuits?

YAEL KALAI: OK. Yeah. Yeah. Yeah. OK, good. Good. Good. Good. OK. Yeah, we're thinking of-- usually in cryptography, we always allow our adversaries to be non-uniform, because they may get advice. So usually, we model them as polynomial size circuits. This is opposed to saying p star that runs in time poly lambda. So we say has size poly lambda. It's just to say he has-- it's another word say, he runs in time poly lambda, but he may have advice.

We allow him to have some kind of fixed advice. That's what we say size. Yeah. Great question. Yes.

AUDIENCE: But if lambda is less than the size of x, how do you even fit in the x bits?

YAEL KALAI: Good. OK, so OK. The problem is actually, yeah. The problem is we don't-- OK, what you're saying is, look. So that's why I have my red pen coming up, my red chalk. You're saying, if x is smaller than, let's say, lambda to the epsilon, if x is of size epsilon we allow p star to be of size poly. So what you're saying is I'm saying, wow. Wait. That's weird. You're not even allowing that if lambda is smaller than x, p star can't even read x. He can't even read it. His runtime is less than x.

True if x is bigger than poly lambda, any poly lambda. That's first, because he can't run in any time. We allow p star to runs in time any poly lambda. So x can be of size any kind of if you take a sequence of x's that are each of size kind of for any lambda, if you have x sub la-- so I'm a cheating prover.

And for any lambda, I have x sub lambda that I'm going to cheat on. And as long as these x sub lambda are at most lambda to the 100, I can run time lambda to the 100 here. This is theory land, so I can do that. But sometimes, we want to actually have communication complexity, which is, like, poly log n. So now the security parameter needs to be poly log n. And now, you can't even read.

So the reason I wrote size poly lambda is because this will allow us to have-- so the communication grows polynomial with lambda. If I allow the prover to run at most poly lambda, it will allow me to rely the-- I can assume that my cryptography, whatever I use here, collision-resistant hash function, or one-way function, whatever cryptographic assumption I use, the assumption is a polynomial time, a polynomial size, cheater cannot break it.

But I can let me strengthen the assumption. I'm going to say that it-- so I'm going to say for every p star of size 2 to the lambda to the o of 1, as long as your running time at most 2 to the lambda to epsilon to find some epsilon, you can't break. Now, x can be like a poly log n lambda.

The thing is now, I'm assuming that a p star that runs in time, this cannot break my cryptography with security parameter lambda. It's a stronger assumption. So there is a delicate issue here with soundness, which I glossed over in the definition above, where I said, computationally bounded. Well, what's the bound?

So again, the reason these computationally-sound proofs are useful is because then, we can use cryptography. So now, we say, let's use cryptography in our proof system. When we use cryptography, there's a security parameter, lambda. Now, typically, we assume that, so now, we want to say, our cryptography cannot be broken. And therefore, our scheme is secure. OK, cannot be broken in what time? So typically, we would like to say, oh in time poly lambda cannot be broken.

That's kind of our polynomial assumption. That's kind of what we think of standard assumptions in cryptography, that some assumption with security parameter lambda cannot be broken time poly lambda. We call that a polynomial assumption. This is what we want. But then, you're right to say, well, it's weird, because then, if you want to think of lambda as being, like, poly log n, if you want, then they want succinct proofs. So let's say we have x of size n.

And suppose we want the succinct to be like poly log n, like in GKR. So now, lambda needs to be of size poly log n in a sense. So it's like, wait, the cheating prover can't even read x. What a weird kind of-- so I'm-- the honest prover runs in time more than the cheating prover? That's bizarre. So the answer is, OK, so the answer is look. It depends on how much risk you're willing to take. What do I mean, how much risk?

If you want to say, look, I'm very, very conservative. I'm not willing to assume-- my trust in cryptography is very limited, and I'm only willing to assume if I use a lambda bit of security, I want my assumption is that a poly lambda cannot break it. I don't want to assume that a 2-to-the-lambda person cannot break it, 2 to the lambda to the epsilon. No, maybe he can. I don't trust that assumption. That's all I'm willing to assume.

Now, but here's the thing. Once you bound the cheating prover to be poly lambda, essentially, what we're saying is that in the world, there's no way you can run time more than poly lambda. If you can, then this has no meaning in a sense. So now, we're saying, look. In this world, you can't run more than this time. Whatever it is, this, this. Otherwise, this computationally bounded is, like, a bit meaningless.

So if that's all the time you can run in, the x's are-- can't be bigger than that, in a sense, because nobody can run at that time. Nobody can read that x. So essentially, what you're saying is you're saying, look. I'm telling you, I want a cheating prover that runs in time at most 2 to the 500. You're saying, well, what if there's an input of length 2 to the 1,000? OK, look. Whatever. He still can run a time when most 2 to the 500. The fact that I'm not going to give him more runtime, it's ridiculous.

I still want to say, for every x. But I don't want to give now the prover more runtime. No. In this world, provers run in time at most some security. This is about the world, about the security of the world. I give a bound on what I think is a reasonable runtime of real-world adversaries. Nobody can run in more time. OK, now you're saying, well, it's weird. You're giving him less time than-- what if for an x that is size 2 to the 1,000, that honest prover, well, there is no honest prover that will prove something about x of size 2 the 1,000.

So anyway we bound, and even if they're, like, x's that are bigger than this, yeah, our pool can't run in time bigger than this. Still, our poor guy needs to prove something about x that he cannot even read. Fine. The point is, in our world, these are our adversaries. That's how we should think of it. OK. Yeah.

AUDIENCE: So the reason we're giving the honest prover this extra poly of n, is that we're assuming that they have to do something with the witness? Is that the--

YAEL KALAI: Exactly. So the prover, the honest prover, of course, he needs to read the witness. Essentially, he needs to look at the input, he needs to read the witness, he'll need to generate a PCP, or whatever. He needs to do something. So the honest prover, of course, he needs to run in time. That depends on x. But the point is you should think, OK, what if honest prover works with x and that is the size 2 to the 1,000. Yeah, but it doesn't exist. It's a theory. It's a mental experiment. We don't have these things. But yeah. He would run.

And now, you're saying our adversary? No, he can't run because our adversaries don't have that power. Nor do our honest prover. They don't exist for that sizes. Yeah.

AUDIENCE: [INAUDIBLE] so before, when we were talking about the information theoretic, ips, and the soundness, I mentioned that as long as the soundness is less than a half or something, you can just repeat it over and over to get better and better soundness.

YAEL KALAI: Yeah. And your--

AUDIENCE: [INAUDIBLE]

YAEL KALAI: Good. Yes. So Yes. The same applies here. So you can ask in a sense, why is here-- so OK. Let me-- it's yes. But we need to be careful. In a interactive argument, you can-- if the soundness here is only half, you can reduce soundness by repetition exponentially. However, you will need to do it sequentially. OK, so you'll need to do one interactive protocol after the other after the other after the other.

Actually, it was a open question for a while was if you have an interactive argument, and you just-- because people don't like-- people care about round complexity. They don't like protocols that take a gazillion rounds. So they say, well, why don't you repeat interactive-- if you take an interactive proof and you repeat it in parallel many times with fresh randomness, everything fresh, just independently, but in parallel, the soundness go down exponentially like you want.

Now, you can say, does the same thing hold for argument? So this was actually an open problem for a while. And then at some point-- it was a long time ago. It was when [Name] was a postdoc, probably 15 years ago. He actually gave an example of, oh, sorry. No, no, no. There was-- sorry.

So there was an example. I'm messing up the-- Baioumy, Meo, and two other people. I don't remember now. They showed actually, no. There is an example of an interactive argument that if you repeat it in parallel, soundness doesn't go down at all. So you have to repeat it sequentially.

So when you repeat interactive arguments in parallel, the crypto doesn't work in parallel as-- it's not really true. It's like, they're contrived examples where you can show that the crypto soundness doesn't reduce like you would hope. And therefore, the soundness doesn't go down. Now, the examples that we have are pretty contrived. OK? It's not like, but there's no-- definitely, it's not true in general. It's not true in general that if you take an interactive argument and you repeat it in parallel with fresh randomness, many times, soundness does not go down.

Even though if [Tak?] showed that there is a way to repeat it in parallel, but you need to be a little clever. Don't just repeat-- a very weird trick he had of, with some probability, you abort. The verifier says, you know what-- I don't want to continue in this. Just a weird abort. He added a weird abort condition, and then all of a sudden, things kind of the soundness vanished. So OK.

So I guess my answer to your question. If you repeat it in sequentially, yes. Air goes down exponentially exactly like we would hope. It goes from s to s to the k if you repeat it k times. If you repeat it in parallel naively, no in their counterexamples. Yet, there are ways to repeat in parallel a little less naively with some abort kind of thing that then it does go down. So I guess the short answer, it's complicated.

AUDIENCE: OK. Thank you.

YAEL KALAI: OK, great. But typically, we just write negligible because of this issue. It's like we like to just, we like to write-- when we talk in cryptography, we usually we write, 1 minus negligible, or 1 for completeness, and then negligible for soundness. OK, so this is the theorem. Any questions about-- I need to define what a collision-resistant hash function is. But any questions before I define? OK, so by the way, just so I know who I'm talking to, how many people know what a collision-resistant hash function is? OK, a lot.

But it's actually not important. And I'll define it in a minute. OK. So let me just define. So a collision-resistant hash family, it's actually a family, it's not a function, h, consists of two algorithms. OK, first is Gen. Gen is an algorithm that generates a key to the hash function. So Gen takes his input. So Gen is a PPT, probabilistic polynomial time algorithm. It takes as input 1 to the security parameter.

So in crypto, we have this hack that we assume our algorithms take as input 1 to the security parameter instead of security parameter, which is it's a bit like I feel like a moron saying 1 to the security parameter. The only reason we write it this way is because we want to think of our algorithm as efficient poly time. And poly is in the input. And all our algorithms run in time poly in the security parameter.

So to say that our algorithms are efficient, our poly time, we always write the security parameter in unary because that's the input. So it's just kind of a hack to make the theory kind of nice. OK? And it outputs a hash key. So this hash key is a size poly lambda. Yeah, because this algorithm is polynomial, it takes an input lambda in unary. So it can output something only a size poly lambda. And then there's the eval algorithm.

And the eval algorithm takes as input-- it's a poly time algorithm that takes as input hash key and x, any x, in 01 star. And it outputs, you can also restrict. You can restrict the x to be, let's say, you can restrict it. But let's think of any input. And outputs a value v in 01 to the lambda. OK, what's important? The way hash functions are used, what's important about hash functions is that they're shrinking.

So you can think it takes an input any x, you can think of that it takes as input like something of size 2 lambda, like takes 2 lambda 2 lambda. You can even think lambda plus 1 2 lambda if you want. But so there's many ways people define it. Let me go with star any input. OK, and now, the security says that a polynomial time, or a poly size algorithm that is given a hash key should not be able to find two different x's that hash to the same value. That's the assumption. Yeah.

AUDIENCE: Can the hash key be an empty string?

YAEL KALAI: What?

AUDIENCE: The hash key, can it be empty string?

YAEL KALAI: It can be the empty string. But then, there's no-- it won't be secure. And I'll explain why. So it's a good question. It's a good question. The question, do we really need the hash key? Or can we just make that empty? So the answer is a bit-- it's again, it's a modeling answer. But I'll get to it. I'll answer your question in a second. Let me just write the security. And then we'll talk about it because it is a great question.

OK, the security, the collision-resistant requirement, says that for any poly lambda size adversary A, the probability that A, given hash key, where hash key is chosen by Gen, the probability that it outputs x and x prime such that x is different, not the same, and eval of hash key and x is equal to eval of hash key and x prime is negligible.

So for every poly size A, there exists a negligible function mu such that for every lambda security parameter, so maybe I'll write it better. For any poly size. For any poly size A, there exists some negative function.

So a function that vanishes faster than any polynomial such that for any security parameter, if you generate a hash key using the security parameter, you give the adversary the hash key, his goal is to output two different inputs that hash to the same value. The probability that he does this is at most negligible in lambda. So as lambda grows big, this becomes tiny. Yeah.

AUDIENCE: Is A's size meant to be polynomial in the hash key?

YAEL KALAI: Good. Yes. Exactly. When I say poly size, I mean poly in the input.

AUDIENCE: OK.

YAEL KALAI: So poly lambda.

AUDIENCE: OK.

YAEL KALAI: OK, that's the definition. Now I want to first answer your question, which is, do we really need a hash key? Can we just tell him, this is the security parameter? Why do we need a hash key? And the answer is, actually, we don't-- it's interesting. We don't quite need a hash key. And actually, the hash functions we use in practice don't have a hash key in a sense. Well, they're fixed. OK, practice is a different story.

But the reason why we have a hash key is we allow-- it's a question of modeling. Who is our adversary? What kind of adversaries are we thinking about? If we think about adversaries that are poly size, that are circuits, or Turing machines that have some non-uniform advice, then what if the adversary just has in his belly, somehow, someone gave him advice, x and x prime that are different and hash to the same value? Someone gave him two x's that hash to the same value. Then, he'll just [INAUDIBLE] it. Of course he has collisions.

I mean, collisions exist by pigeonhole principle. I mean, huge, yeah? It goes to small. These exist. We want to say they're hard to find. But if we think of a non-uniform adversary that has advice, he may have this as advice and then he'll output it. So if we think of a non-uniform model, then we have to have a hash key. Another option is to think of a uniform model. Let's think of A not as poly sized, poly time. He's uniform. Then we don't need to have a hash key.

We just, because we can assume that small axes don't collide because we'll just make the hash function so they won't collide. And then big ones, you can't hold in your belly because you're bounded. You're uniform. You have only constant sizes kind of your constant size description. So actually, there have been works in the literature in the theoretical cryptography literature that talk about hash functions with no keys.

But then, they need to talk about uniform adversaries. And this is something that is considered and talked about. But the most standard is to talk about non-uniform adversaries. And then, we need to have keys. Yeah? OK. Any questions? OK. So oh, our time's up in five minutes. So good.

So now, what I want to do is I want to prove this theorem and I want to give you a construction of a hash function that we use. Now, let me tell you, actually, we use a hash function with an additional property. For this theorem, we use a hash function with an additional property. We don't need to assume it because we can construct it from a hash function. So that's why I didn't write it as an assumption, because I can construct it.

But really, the primitive we need for the theorem is we want an additional-- we want a hash function with what we call local opening. What do we mean by local opening? Here, a hash function, we say, look. There's an eval that takes a hash key x, output a value. Now, if you want to check, I'll give you-- let's say I hashed a value. I told you, here's v. Now, you're saying, OK, I want to know what you committed to. Fine. I can give you my x. And what you hash-- sorry. Which value you hashed?

I'm like, no problem. Here's my x, and you can check. But what I really want for this theorem, and it's something I want from, this is a common primitive that we need in many applications, is I want to say, OK, I hashed-- I took a hash function. I hashed a value. I want to argue, look, I hashed a value. I don't want to give you my entire value. It's too big. You guys are not the verifiers. You can't even read everything I wrote in my-- I can't give you what I hashed. It's too big.

I want to open one bit. I'm going to tell you, look, I hashed a value. But the ith bit of the value I hashed is 0. How do I know? How do you know that it's 0? Oh, I'm going to convince you. I can kind of give kind of a local opening to this x. This is what I want. So what I defined here is just a hash function. But then, I can define another primitive, which is a hash function with local opening. And that's what we actually need for the theorem.

So a hash function with local opening has two additional algorithms corresponding to it. There is an open algorithm, which allows you kind of to open to show kind of a, essentially give you a little proof that the ith bit of what I hashed is a certain value. And then there's a verify algorithm that verifies it.

OK, since we have only two minutes, I think I'll skip-- next time, I'll show you the local opening. But maybe I'll just tell you one minute of the idea behind-- is my theorem hiding somewhere? Maybe here? Oh yeah. Sorry. It's the opposite. OK, behind this theorem. So the idea is what I want to do is the idea at a high level, what they do, they take, the prover will take this ginormous PCP. He wants to give it to the verifier, but the verifier can't hold it. So what are you going to do? He's going to give him a hash of this PCP.

Take this PCP and just give him the hash value. Just slam the bits. Now the verifier has this hash value. Now, in some sense, now, the prover-- now, the hash value, now the verifier is going to tell him, OK, thank you very much. I'm going to behave as a PCP verifier. So I'm going to generate randomness, ask to look at locations i1, i2, up to il in your PCP. Give me an opening. Show me. Show open--

Don't give me the entire PCP. I can't hold it. Give me only location i1, i2, up to il. Now, the prover-- once again, here, no problem. It's b1, b2, up to bl. But the verifier needs to be-- how does he know that it's indeed the opening of the hash? So we're going to use a hash function that you can open. So he's going to give him the bits with openings. See? Here's an opening, or a little proof, that in location i1 is sitting b1, location I2, I have B2, and so on and so forth.

And then the verifier is just going to check the opening that they're good and accept if the PCP verifier accepts. So essentially, all the idea, you combine this PCP with kind of this cryptography technology that kind of smushes this PCP into a small thing, and allows you to kind of look kind of only at-- allows the prover, in a sense, to look only at specific locations computationally. Like, you can give him little proofs that open specific locations.

And that's it. That's what you get. So next time, we're going to define the local opening, we're going to see the interactive argument in detail, and prove that it satisfies the guarantees written here. OK, so we'll see you next Friday. Thanks.