

[SQUEAKING]

[RUSTLING]

[CLICKING]

Yael Kalai: So last week was crypto day, so I didn't get to see you. So let's do a quick kind of recap of where we were and where we're heading to. So last time I saw you, at least in this room, was two weeks ago, where we talked about how to use the GKR protocol to construct what's called a PCP, a Probabilistically Checkable Proof.

And what's a probabilistically checkable proof? The idea is, take a proof-- so take any NP language, let's say, that has a proof, like, let's say, 3-SAT-- you take a witness or a proof, so a satisfying assignment, and you actually make it longer. OK, so you don't shrink it. You actually extend it. It's bigger.

But now, to verify it, you don't need to read the entire proof. What you need to do is just read a few locations in this proof so you can-- even though the proof actually became longer, you can verify it much more efficiently by reading just a few locations. And we actually showed how to construct it, How do you get a PCP? And the idea of how to get a PCP, the idea was, well, let's look at the GKR protocol for 3-SAT-- so let's do a PCP for NP.

OK. So any NP, you can convert it to a 3-SAT. You know 3-SAT is NP complete, so convert whatever you want to prove into a 3-SAT formula. Now you have a witness.

Now what do you do? You want to generate a PCP. What you do is you pretend you're a GKR prover that has this witness, the satisfying assignment, and want to convince the verifier that this is a valid witness.

So what do I mean by run the GKR protocol? What I mean is-- let's look at this. So given ϕ , let's say, a 3-SAT instance, what you do, you look at the circuit C sub ϕ , that takes as input a satisfying assignment, an assignment, and outputs 0 or 1 whether it's satisfying or not.

This is a very low-depth circuit. It's just as kind of a bunch of checks and adds the local checks, and adds them all together. And now what we do is-- how do you generate the PCP given this witness, given this witness for ϕ ? What you do is you run in your head the GKR protocol for the circuit.

So remember, the GKR protocol is a protocol that proves that a low-depth circuit-- that an input is-- that a given input-- even though it's weird, the verifier doesn't know the input, so it's kind of weird to think of GKR in this context. GKR, the context of GKR was there's an input, everybody knows the input, the prover and the verifier, but the circuit is large and the prover wants to convince the verifier that the output is something.

Here, it's kind of weird. The input is more or less as large as the circuit, OK, because checking if a witness is satisfying the assignment is valid for a 3-CNF. The 3-CNF is-- the circuit here is almost the same size as the witness.

And so it seems like using GKR here is very weird. It doesn't seem like the right application. And in particular, the verifier doesn't even have the input, so how can you use GKR? That seems very counterintuitive.

So the idea is, no, let's pretend that he does have the witness. And what do you mean, pretend? Oh, don't worry, we'll give it to him.

So in the PCP, we'll write the witness. Not only we'll write the witness, we're going to actually write the low-degree extension of the witness in the PCP. OK, so what is your big proof? It first contains the low-degree extension of the witness sitting there.

Then, now think of the GKR protocol. For each and every kind of question of the verifier, the prover has an answer. Write it down in the PCP.

So in the PCP, write, if the first message of the verifier-- you know, the first message is a bunch of-- well, first, the approval goes first, so you send your univariate polynomial. Then the verifier sends a field element.

Now we say, well, if the field element was 1, here's my answer. If the field element was 2, here's my answer. For each and every possible field element, you say the answer.

OK. Now, if there's only polylog number of field elements, it's not so bad. OK. Then you give them the answer. Now the prover sends another field element. Now you say, for any possible next message, here's my answer, and so on.

So, essentially, you kind of open up this GKR. It's a gigantic tree. Now, first, you say, wait, this is huge. Like, it's exponential. But actually, GKR has a very nice property that the prover's messages actually only depend on the last very few messages of the verifier.

So if you remember the GKR, it's a bunch of sum-checks protocol. It's, like, just a sequence of sum checks, and the prover's message depends only on the messages in the last sum check. So it's kind of very few, and kind of where you start with, it's very few messages.

So to be a little more precise, in GKR, each prover message depends on the last kind of-- what it is, it was " $\log n$ over $\log \log n$ " messages from V , from the verifier. Well, this is like the number of variables. If you remember, when we did GKR, it was a bunch of sum checks, and each sum check was over m or $3m$ variables.

This is the m . And the field is polylog n . And here is the size of the circuit. So think of it as, the size of the witness, the size of the 3-CNF, it's all up to polynomial factors. It doesn't matter.

And so now, OK, you need to depend on all these messages from the verifier, but there's only that many each time. And if you look, this to the power of this is only poly. That's exactly how we chose H and n so it would be poly. That's why we chose this weird setting of parameters and not just $2n$ and $\log n$.

But the point is, because GKR has this kind of memoryless property of the prover, you can expand all the GKR into a big proof of size poly n . Now what does the verifier do? The verifier simply-- so that's what the PCP is. The PCP, you write for every possible kind of-- you write all the possible answers of the provers and the low-degree extension of your witness. That's what the PCP is.

How do you verify this PCP? Well, you pretend you're a GKR proof verifier. You say, OK, let me see your first question, thank you.

Now you're saying, I'm going to choose a random field element. And I'm going to look at the answer corresponding to this field element, and then I'm going to choose my next random field element. And you just go through this kind of tree randomly and make sure-- and at the end, you need to verify.

If you remember, in GKR, at the end, you need to verify. To verify, you need to check your input and a random point in the low-degree extension. This is how GKR works. Well, you have the low-degree extension sitting there as part of your PCP, so you check that.

OK, that's basically it, except there's one component that we're missing. And that's going to appear in your pset, which, by the way, is going to be out this weekend. OK, so this weekend, the pset is going to be out.

Originally, the due date was after Thanksgiving. I'll move it to before, just to give you an incentive to not take it with you on your break. If anybody needs an extension, just let me know.

OK. So you'll have almost a month? No, not quite, but close. OK. But if you need an extension, just let me know. This is very arbitrary. OK.

OK. So what's the problem? The problem is, so if indeed the-- so what about soundness? OK. Soundness seems like it should follow from just GKR. OK. You're interacting with a prover, a GKR prover, and some should just follow from the GKR, except that there's one point here which is, What if the PCP prover doesn't give you a low-degree extension of some input?

So if he gave you a low-degree extension of an input, and this input did not evaluate to 1-- namely, if he's not satisfying assignment-- then this input is not a valid witness, because one does not exist. In that case, of course, the soundness would really follow from GKR soundness.

However, one needs to think, there may be a malicious prover who doesn't give you here a low-degree extension of some W . So what he should give you-- a low-degree extension of W , it means, like, some W goes from F to the m to F , and it's of degree H minus 1 in each variable. OK, if it's of degree h minus 1 in each variable, then it extends, it's a low-degree extension of kind of the W sitting kind of in-- so then W is just W sitting on H to the m . That's kind of the little cube where W is sitting.

If, indeed, he gave you a low-degree extension of some W , then you're good. If this W that he gives you is of degree H minus 1 in each variable, then you're good. But what if it's high degree?

What if this W -- what if he gives you some W star which is very high degree? Then is it sound? So in the homework, you'll have exactly this. Actually, it's not sound, and you'll have to find, actually, an attack.

And then there's a way to make sure. So now what the verifier needs to do in addition is to do what is called a low-degree test. It's a test that verifies that this W star has low degree. OK, so this is what the homework assignment, or one part of the homework assignment, is about.

OK. But this is-- imagine we're making sure that this is-- actually, the verifier can't make sure it's low degree, but it can make sure it's close to a low degree. So it agrees with some low-degree W on, like, $1 - \epsilon$ fraction of the points. So you can test that it's very close to low degree.

If it's very close to low degree, then, most likely, wherever you'll ask-- because you'll ask randomly-- it agrees with kind of the low degree, and therefore, that's kind of the idea. OK. But this is the PCP, and the verification just adds a low-degree test. And we'll expand on that in the pset.

OK. So that's kind of what we did most of the class, and then at the end, we talked about-- OK, so now we have a gigantic PCP. It's very large. But you can verify it by looking at very few locations, which is nice.

How many locations? Like the communication complexity of the GKR, which on circuits of depth $\log n$ or order $\log n$ -- which is, for example, 3-SAT-- then the communication complexity is $\text{polylog } n$. So you can verify this by reading only $\text{polylog-}n$ locations, as opposed to n locations, so it's, like, an exponential improvement. But again, you have this gigantic PCP.

And now the question is, OK, but who's going to-- note, here's something to just make sure you understand. It's very important that this PCP is kind of given to the verifier before-- if the verifier-- so, OK, the PCP is very large.

Now, one can say, OK, don't send the PCP. I'll tell you what my queries are. Just give me the answers to my queries instead of sending me-- anyway, I'm not going to read them, so why send it over? Just send me the answers to my queries.

That will not be sound. If the verifier, if the prover gets to choose the answers based on all the queries that he sees, soundness breaks down completely. OK. And so it's very important that he first tells you what the PCP is and then you query it.

OK. In particular, the damaging-- the way the cheating prover can cheat is by-- you know, when you tell him, give me the PCP in location i , if you ask for it together with location j , it will give you one answer. But if you ask it for location j prime, it will give you another answer. That's how it can cheat. But if he needs to fix it ahead of time, then, no, it's one answer-- doesn't matter the other locations you ask-- and that's what makes this sound.

OK. And you can actually go back to GKR and see, if you use this PCP with GKR, where you give the queries ahead of time, you give the sum-check queries all ahead of time, everything breaks down completely. OK. So it's very important that you first commit it to this entire PCP.

And then one can ask, well, if you give this-- how does the-- so the entire thing says, look, the verifier is a weak device. That's the idea. He's weak. He cannot do the computation.

How can we store this gigantic PCP? So it seems like, well, that's very nice, but how do we use this thing? And the way we use this thing, the way it's used today-- these things are used today, and the way they're used is using cryptography.

So what we do, we use cryptography to take this gigantic proof and squish it. OK. And this is used using what's called collision resistant hash functions. So this is what we're going to kind of-- we're going to talk about today. We started last time, but we're going to continue today.

But before we go into how we squish it and make kind of everything succinct, let me first-- once we use cryptography, we can no longer get soundness against any all-bounded kind of cheating prover, because cryptography can be broken by all-powerful adversaries. Cryptography assumes the adversary is not strong enough to break our assumptions-- for example, the collisions and hash function that we saw, and we'll review it in a second.

So now, when we talk about proof systems, we need to relax the notion of what we mean by a proof system. By the way, who's scribing today? Ah. OK, great.

We need to relax the notion of what we mean by a proof system to what's called a computationally sound proof system. Namely, you cannot cheat, assuming you're polynomially bounded. If you're all powerful, you can break my crypto-- no guarantees. But as long as you cannot break my crypto, you cannot cheat.

So I just wrote it here. This is copied from last time. So instead of interactive proof, we called it an interactive argument, and we say that an interactive argument for some language L -- think of it, let's say, NP language, but whatever language you want to think about-- is a protocol between a proof and a verifier.

The completeness is the usual. Namely, if x is in the language, then you'll convince the verifier with probability 1. OK. And if it's an NP and you know a witness, you're also efficient. OK. The soundness, that's where the change is. Instead of soundness, you get some computational version of soundness, and the computational version of soundness says the following.

It says there's some parameter, a security parameter-- when you're talking cryptography, there's always a security parameter-- which kind of says, How much time do we believe real-world adversaries can run in? OK. So what the soundness says, that any real-world cheating prover-- real-world meaning it runs in time polynomial in the security parameter.

So we're going to denote the security parameter by λ throughout this class. Sometimes they call it by n , and sometimes k . Here, n is going to be input size, so I don't want to-- so we're just going to use λ .

So for any P star of size polynomial in the security parameter, he can win only with negligible probability. Namely, there exists a negligible function μ -- again, negligible means it's smaller than any inverse polynomial-- such that no matter which security parameter you use, and any x -- so think about the P star. He's polynomial in the security parameter. He chooses an x adversarially.

The probability-- or, an x is chosen, OK. It exists. For any x , the probability that he convinces-- now, the protocol dependent on the security parameter, so the verifier, may choose, like, a hash function, according to the security parameter.

So everybody knows the security parameter. Everyone runs in time polynomial in the security parameter. And that's why we give it as input, the security parameter, in unary.

So an input x and security parameter-- there's a protocol, and the probability that this cheating prover convinces the verifier to accept is this negligible function μ . OK. So again, we offer soundness only for poly- λ cheating provers. OK. So now you should think of the cheating provers as like a family of provers, one for every λ .

OK. It's like a friendly security parameter. You're allowed to give a cheating prover. But they all, let's say, run in time at most λ to the 10, OK, so polynomial in λ . Any questions about the review before we go on to the new-- to today? Yeah?

AUDIENCE: Just, again, so here, the proof is not showing you, like, hide the witness or x [INAUDIBLE]?

Yael Kalai: No. No, no, no. Good, good, good. So, two things-- first of all, we're not at all talking about hiding. If we didn't want succinctness, you can tell the proof will just give you the witness. Here you go. Check on your own. OK, but we'll want another property from these interactive arguments, which is succinctness, yeah, succinctness or-- yeah, exactly, for NP, for example. And-- yeah?

Audience: So 3-SAT, I take it that means Boolean formulas?

Yael Kalai: Yeah, exactly. 3-SAT is just-- think of it, it's-- exactly, just three CNFs. It's the AND of a bunch of-- of a bunch of clauses. Each one is like, x_1 OR x_2 OR not x_3 .

Audience: And do you get it with a small enough depth?

Yael Kalai: Yeah, so any 3-CNF, any 3-CNF is of the form, like, clause 1 AND clause 2 AND clause n , let's say. You can get a Boolean formula with fan-in 2 of size order $\log m$, because you just do kind of the AND, AND, you know, all the ANDs. And then you have these little literals at the bottom, and each one is constant size. It just depends.

Each clause is constant size. So it's really a $\log m$ circuit. OK, so you can take any 3-SAT, which is just an AND of a bunch of literals, and write it as a \log -depth circuit.

Audience: So can you go directly to some j and do the PCP over some j by doing the parameterization instead going through GKR?

Yael Kalai: OK. To get a PCP, I need to go-- the way I construct the PCP is by going through GKR. I do a GKR on this kind of circuit, I open up all the possible provers' messages, and that's what constitutes the PCP, in addition to the low-degree extension of the input. That's going to be the PCP.

And the verifier just kind of verifies, behaves like the verifier, and checks that the witness he gets, the extended witness, is of low degree. Like, it does a low-degree test. The low-degree test, we didn't talk about, but it's going to be in the homework.

So that's how the PCP works. And now I'm kind of talking about how you use this long PCP together with cryptography to get these succinct arguments. That's kind of the next step, How do you use cryptography to just have a fully succinct protocol?

Let's say, for NP, we want a protocol that has polylog communication. OK, so you have an instance of length n , and the prover wants to prove to the verifier that x is in the language, and you want the communication to be only polylog n , for example. OK. OK, one thing I do-- oh, did you have a question?

Audience: Is there just a quick-- [INAUDIBLE] is the interactive protocol necessary? Because in the traditional complexity series, PCP, probability has to be written down in advance, and it cannot be adapted [INAUDIBLE].

Yael Kalai: Right. Right.

Audience: So is the whole point of the protocol trying to fix some problem in a sense, where you're not forcing the prover to commit to--

Yael Kalai: Exactly. Exactly.

Audience: OK.

Yael Kalai: That's exactly it. That's exactly it. Exactly. One thing I do want to say, note that I asked for the cheating prover to be polynomial in λ , OK, because that's what my assumptions usually look like.

I want to say-- if I use collision resistant, I want to say, well, I used some collision-resistant function with security parameter λ . It says that my assumption is that a poly- λ -sized circuit cannot find collisions. OK. My cryptographic assumption usually assumes that they are hard to break against adversaries that run in time poly λ .

But this may seem very weird and very weak. Why? Because, you know, our goal is to come up with succinct arguments, OK, succinct. In particular, one example to keep in mind is let's say the input n can be really huge, but I want the communication complexity to be polylog n . So think of λ as being like polylog n . OK, everything's going to go with the security parameter, but let's take the security parameter to be polylog n .

Now it seems weird. You're like, Wait, really? So the cheating prover, if the instance is of length n and the security parameter's of length polylog n , the cheating prover, we're not even giving him the opportunity to read the input, so it's a very weak guarantee. So what does this guarantee tell you?

Let's say I want you to prove to me that a given x is in the language. And my only guarantee is, if you run in time that's only polylogarithmic in n , you can't cheat. OK. But what if you run in time n ?

So that seems like a problem, especially if-- look, if n is too big for anyone to read, then in some sense, why-- nobody's going to use it anyway. And if n is-- we're able to read it, if I'm able to read it, then you have the time to run in that time, so you run in time n . I'm not going to assume you run in time polylog n .

So this seems very weird. You should not be happy at this point. You should be like, wait, something is going on. I don't like this definition. It's meaningless.

Like, look, if you don't want succinctness, you can make λ be size of n and then you're happy. If λ is of size x , the instance size, then you can say, yeah, the instance size is some n . I allow my real-- my honest prover runs in time poly n . I allow my malicious prover to run in time poly n , and that makes sense to me.

But now I'm saying, no, the honest prover, he can actually run in time n . The malicious cannot. Like, no, the malicious has more power than the honest.

So it's really a weak guarantee, that kind of-- especially if we want to take the security parameter, the idea is we're going to take the security parameter really small, OK, really, really small. But if we take the security parameter so small, we should give the adversary more power than calling the security parameter, because it just doesn't meet real life-- the idea is to-- the idea of this is to kind of capture a real-life setting, not to make a model that makes no sense. So if we take λ to be very, very small, then actually the real-world adversary can run in time more than poly λ . It can run in time poly n , so it doesn't capture-- yeah?

Audience: If P star's the circuit, how do we even think of plugging in a variable length back into a [INAUDIBLE]?

Yael Kalai: They can't even read x . You're right. Yeah, for example.

Audience: [INAUDIBLE] not even [INAUDIBLE].

Yael Kalai: So you can say, oh, P star can't read x . You're right. They can't read x .

It's a circuit, and, yeah, it can't read x . This is a big problem. OK. So the way I defined it really doesn't make much sense. Yeah?

AUDIENCE: Just to make sure, why did we need λ to be $\text{polylog } n$? Is it so that the resulting interaction is succinct enough?

Yael Kalai: Exactly. So, great question. The question is, Why do we need it to be $\text{polylog } n$? So I think where maybe you're going to-- you know what, let's make λ n to the ϵ , for a constant ϵ , n to the 0.001 .

Then we say, look, the original proof grows with λ , which is n to the power of 0.001 , which is much smaller than n , so we're still happy. But the cheating prover can run time any poly, any polynomial in it, so it's $\text{poly } n$. So that's good. You're right, and that's a good setting to think about. In that case, there is no issues.

So one thing which you're saying is you're saying, let me fix all your problems. Just take λ to be-- think of it as n to the 0.01 and you're done, or n to the ϵ and you're done. OK, if you think about n to the ϵ , everything kind of makes sense. You allow the cheating prover to run time $\text{poly } n$, the instance is n , and we're good.

OK. So that's one way to think about it. So usually, the way we think about it, we say, for any n , we choose a security parameter that depends on n . OK, that's usually how we do it. We have an n , and now we say, OK, which security do we use?

It's kind of a function of n . So that's one example. Choose that to be perfect, and then you've solved all the problems.

But what if I want more succinctness than that? Maybe n to the 0.01 is not-- it's OK, but I want better. I want $\text{polylog } n$. So what if my λ -- that's one option, but another option is $\text{polylog } n$. That's even more succinct.

So that's what I want. Now you're saying, well, if that's what you want, you've got a definition that makes no sense. And you're right. But then what I can do is strengthen the definition.

So how do I strengthen the definition? I'm going to say, you know what, poly is one option, but let me actually assume, make a stronger cryptographic assumption. I'm going to assume some λ the same for any adversary that runs-- for any P^* that executes a circuit not of size $\text{poly } \lambda$ but of size 2^{λ^ϵ} to the ϵ .

I'm going to allow my adversary to run in time 2^{λ^ϵ} , more than poly. Now it's a stronger assumption. So now I want to say, an adversary that runs in time 2^{λ^ϵ} or of size 2^{λ^ϵ} cannot break my hash function. He cannot find collisions. It's a stronger assumption.

But now this makes sense. So I can take λ very, very small, but I'm saying, even if the adversary runs in time that's much bigger than λ , that's actually 2^{λ^ϵ} , even he cannot break my crypto. And that's why, when we talk about succinct proofs, most often, we talk about this kind of assumption.

Subexponential assumptions come up a lot, and we'll see that, actually. But often, if we want to implement, so if we want to implement, if we want to take λ to be $\text{polylog } n$, it makes sense to require-- for the soundness to make sense, to correspond to real-world applications, it makes sense to require subexponential assumptions of our underlying cryptography.

OK, so just to kind of close these parentheses, one can talk about poly-size cheating provers. It will give you a more standard assumption. But the soundness guarantee you get may not be strong enough, depending on how security parameter relates to the input size. If you want λ to be very, very-- the security parameter to be very, very small, and yet you want to offer security against adversaries that runtime poly n , then you will need to make your assumptions stronger.

OK. But this is kind of a design choice. OK. For now, I'm just going to think of, for any given P star of size poly λ or more, whatever we choose here, that corresponds to the assumption. OK. So if we have T secure assumption, then for any T size-- so in general, more generally, you can replace this by T of λ , right, any T . It can be poly. It can be subexponential. It can be whatever.

OK. And then we choose the T , and now the T kind of governs the application. But once you have a T , now we can just prove it using math. We don't need to think about it anymore.

OK. So that's kind of how λ relates to n and how the strength of the assumption relates to n . OK. Any questions about that?

OK. So now let's go ahead with how we use it. So that's what we want. I want to come up with a succinct, interactive argument for all of NP. OK. I want to show how I can take any NP language-- and let's think of 3-SAT.

Why not? It's complete. I want to prove to you that a 3-CNF formula is satisfiable. I have a witness of length n . I want to prove to you that it's satisfiable.

But with the communication, I want it to grow only with λ . OK, λ can be polylog n . I don't even want to-- don't go now to the choice of how we chose the λ . But now I have a security parameter λ , I want the communication to only go with the security parameter, not with n .

OK. So how do I do it? So the idea is, well, I'll use my PCP. I have my PCP. I'm going to use crypto to squeeze it.

How do I use crypto to squeeze it? It's exactly what collision-resistant hash functions do. They take a big thing and they squish them down.

So now I'll take I'll take my PCP, the prover will take his PCP, he will squish it down using a collision resistant hash function. OK. So he gives the verifier-- here you go. Here's a-- like, a squished, like a digest of this PCP.

The verifier takes this thing. It has no meaning. So what does he do? Now he says to the prover, OK, yeah, thank you very much, but I want to see the PCP on a few locations.

So now what can the prover do? The prover can't-- OK, so collision-resistant hash function, as is, it's a way to kind of compress in a way that you can't-- you're kind of committed, because you can't find collisions later. So you can't open-- after you gave the hash value, you can't open to x and x prime. You can't give x and x prime that both hash to the same value.

But I don't want you to give-- you can't give me-- now you're hashing an entire PCP. The PCP is big. You can't just-- I never want you to give me-- me, the verifier, I never want to receive the PCP. I can't even hold it. I want to receive a few bits of the input.

So what we need is a collision-resistant hash that has a special property that you can open bit by bit. So instead of giving me the entire input, the entire preimage of the hash value, I want you just to give me a few bits here, a few bits here, and a few bits here, and prove to me that they correspond to the input you hashed.

So now I'm going to define this primitive. Once I have this, we're going to see it's going to be very easy to come up with the actual interactive argument. OK. So the primitive is what's called a collision-resistant hash with local opening. Let me define it.

So we're going to have a collision-resistant hash function with local opening, and this consists of five algorithms. The first one is-- the first two are Gen and Eval. These are exactly-- these two algorithms is what defines collision-resistant hash function.

So this, we saw last time. I'll just repeat, just to recall, Gen takes as input a 1 to the λ , the security parameter, and outputs hash key. So this is some key.

Eval takes as input hash key and any input x from $0, 1^*$, anything, OK, any string of any length, and it outputs a hash value. We'll denote it V . And that's fixed at $0, 1$ to the λ . I call it sometimes δ and sometimes λ , right? Sorry. It's λ .

You output a hash of a fixed size, which means you can take very, very long things, and you digest them down to a value of size λ . OK, so it takes an input of arbitrary size. It can be huge. It can be 2λ , 4λ , tons, huge, and it digests down to a string of length λ .

OK. Now there's two more algorithms-- two or three? OK. So now there's an algorithm called Open. Open, I just want to make sure-- I want it to tell me-- don't give me the entire-- I want to-- so these just digest. It has no meaning.

I want to tell you now I want the i -th bit of x . Don't give me all of x , what you hashed. I can't hold it. I just want the i -th bit.

So, Open, so what do you do? You open. How do you open? You have the hash key.

You have x , which you computed, the value v . And now you have some i in kind of $0, 1$ to the length of x . So this is kind of which index I want you to open.

And what you output is a bit B -- which is kind of corresponding to x_i , the i -th bit-- a bit and an opening value ρ . OK, this is some kind of string, and this is of size λ , so, small. Or it can be $\text{poly } \lambda$, OK, an opening of size $\text{poly } \lambda$. Yes?

AUDIENCE: In this definition, who's running the Open [INAUDIBLE]?

Yael Kalai: Good. Who's running the Open? Good. The person who generated the hash.

So let's say you're the prover. You want to prove to me that ϕ is satisfiable. You generate a PCP.

I don't want your PCP. It's too big. You give me a hash, a value V . Thank you very much.

Now I want you to open the PCP and some location i . So I tell you, here's i . I don't know your PCP. I don't know your input.

You give me some ρ_i , like an opening-- give me x_i . If you only give me x_i , you can cheat. You can give me whatever you want. I don't believe you.

So I want you to give me kind of-- ρ_i , this opening, is like a proof that indeed what I hashed down in the i -th location is sitting x_i . This is what you think of ρ_i . It's like a proof. Yeah?

AUDIENCE: Shouldn't i be in the log of the [INAUDIBLE]?

Yael Kalai: Oh, sorry. Sorry, sorry, sorry. Sorry. Thank you. Sorry. It's in index, not in 0, 1. Sorry. It's in index, with 1 until the length of x -- 1 to n , let's say. Yeah. Thank you very much. Questions? Yeah?

AUDIENCE: Not so much a question as just, like, I guess it's just interesting that you have to retain the entire hashed value x . Maybe that's necessary for correctness, to be able to confirm any bit of x , but I don't know, it just seems interesting that in practice, you would have to keep storing all of these hashed values.

Yael Kalai: Right, so what you're-- wait, let me make sure what your comment is. Note that to open, you need to know the entire x , and why do you need to know the entire x , in a sense, to open to a bit i ? So in some sense, look, I mean, to open to bit i , all you need to remember is ρ_i . That's what you need to give for the opening.

But all the ρ_i 's together contain all the x . So that's why you need to kind of store all the x . And you'll see the construction soon, so it'll be kind of clear.

OK, the last algorithm is what we call Ver, or Verify, and Verify, it takes as input a hash key of-- and now I'm verifying. You gave me x_i and ρ_i . How do I know it's good?

So I take my hash key, the hash key, the value that you gave me, i , x_i , and ρ_i , and this algorithm outputs 0 or 1-- namely, accept or reject. So I can verify whether the ρ_i is a valid opening.

OK. And now, what are the guarantees? So you can hash, and then you can open. What is the guarantee?

The guarantee is, first of all, if you do everything honestly, I'll accept you. OK, so there's two guarantees. The first guarantee is just correctness, and the second guarantee we want is collision resistance. So we'll see.

So for correctness-- before, actually, let me just-- sorry, let me just say, these are all PPT algorithms. OK, all of them are polynomial time. Actually, the only-- everything is-- these are-- I should say, these are poly-time algorithm, and this is PPT.

So this is a randomized algorithm, poly time, so it runs in time $\text{poly } \lambda$, generates a hash key. These are all deterministic algorithms, OK, that run in poly time.

So Eval takes his input x . He runs in time x , of course. Open runs in time x . But Verify runs in time $\text{poly } \lambda$.

Why? Because the hash key is of size $\text{poly } \lambda$. This is of size $\text{poly } \lambda$, because I required it. And x_i is a bit, and this is of size $\text{poly } \lambda$, because I required it.

So the verifier runs in time $\text{poly } \lambda$. OK, so when you send me a hash value, I get a value of size λ . When I want you to open to a certain bit, I can check it in time $\text{poly } \lambda$ independent of n . I mean, λ may depend on n , but it doesn't grow with n . Yes?

AUDIENCE: Why [INAUDIBLE] ought to be probabilistic?

Yael Kalai: Why is Gen--

Audience: Just Eval.

Yael Kalai: Eval is not-- these are polynomial time.

Audience: Yeah. Why is it not allowed?

Yael Kalai: Ah. Why is it not allowed? It doesn't need to be. It just doesn't need to be probabilistic.

I'm not inherently opposed to using randomness in algorithms, but it just doesn't need to be, whereas Gen has to be probabilistic. OK. Great. Thanks. Good question.

OK. So the correctness guarantee just says that-- let's see-- for every λ , every hash key generated, like, in the image of Gen, and every x -- OK, not every x . We get completeness up to x in $0, 1$ to the 2 to the λ . OK. Or, I should say, sorry, small or equal to 2 to the λ .

So let's think of the security parameter. It's at least of size $\log n$. n is the size of x . OK.

We never take the security parameter to be smaller than $\log n$. OK. And so that's why we always assume that the size of x is at most 2 to the security parameter, the number of bits in x . OK, so for every x , we want the probability that Verify-- oh, I forgot to say, for every i , the probability that Verify hk , Eval hk of x -- this is v -- what else do I have?

i , x_i , and ρ_i , which is Open-- this is kind of the ρ -- it outputs 1 with probability 1 . So if you are honest, you take some input x , any hash key, and you give me a hash value and then you open it, I'll accept you. OK. If everyone's doing what they're supposed to do for any x and for any hash key, we'll get output 1 .

By the way, sometimes, one can define also with high probability over randomly choosing a hash key. But our constructions work often in the worst case. But one can also-- a weaker definition is for a kind of hash key randomly generated.

And here you have 1 minus negligible, where the probability is over the hash key. Here, actually, sorry, here there was no-- OK, before, there were no probabilities, because I just chose for average. But one can think of a hash key randomly generated, and you can have a negligible probability of error. We won't need that because-- I mean, some schemes, but many schemes are just probably 1 . Yeah?

Audience: So the [INAUDIBLE] that correctness always holds for any hash key, but soundness needs the randomness, right?

Yael Kalai: Right. Yeah, I didn't go to soundness yet. Yeah, yeah, yeah, yeah, yeah. We'll see soundness in a minute.

But correctness, we can get for every hash key. But one can think of constructions that actually-- even correctness with high probability. But often, we think of it as correctness 1 . Yes?

Audience: For correctness, you have the position of x ?

Yael Kalai: Again?

Audience: [INAUDIBLE] for correctness, you have that restriction on the length of x , so the length should be less than 2 to the λ , right?

Yael Kalai: Yeah. Yeah, yeah, yeah. You're asking why?

Audience: Yeah.

Yael Kalai: Because our constructions, often, we'll see a construction, and we'll see that if the length is-- as I said, OK, the reason is our construction grows with $\log x$. That's the truth. It just grows with $\log x$, and we want the output to be λ .

So the fact of the matter is, the way we construct these schemes, our construction, the hash value grows with $\log x$. But we don't want to kind of say it. We want to just say λ , because it's nicer.

So we just make sure λ is at least $\log x$, and then we're OK. OK, great question. OK, any other questions, before I go to soundness?

OK. So soundness, that's the more interesting one. Soundness says, intuitively, that for any bounded poly λ , or T bounded, there's some-- the soundness is like-- you can call it T soundness. OK, and you can think of-- poly soundness, you can think of it " 2 to the λ to the ϵ " soundness.

But there's some T soundness, and it says any adversary that runs in time of size at most T , the probability that it can find a value and an index, a hash value and an index, so that at that index, he can successfully open to both 0 and 1 , is negligible. OK, so it essentially means, once the adversary gave you a hash value, for any index, he cannot generate a valid opening both for 0 and for 1 . It's impossible for him.

OK. So let me write it formally. So for any poly T -- again, ϕ of T is, like, linear, so poly λ -- poly- T -sized adversary, the probability-- and the probability is over hash key-- he gets a random hash key chosen by Gen, OK, gets a random hash key chosen by Gen. This adversary, given the hash key, he tries to open some index in two different ways. So his goal is to give you a hash value v , an index i , and a valid opening ρ_0 that opens to 0 and a valid opening ρ_1 that opens to 1 .

That's what he outputs. So the adversary outputs-- he gets a hash key. He outputs a hash value, an index, and two openings for that index-- that's what he's trying to do-- one corresponding to 0 , one corresponding to 1 . And the probability that both of them are a valid opening, the probability that for every b in $\{0, 1\}$, Verify agrees-- so Verify, given hash key v , i , and b and ρ_b , outputs 1 , this is negligible.

OK. So again, soundness is like collision resistance in this setting. So this is kind of another way to-- usually, people call this property collision resistance. And the collision resistance property says that once the adversary chose a hash value, the probability that for any index-- OK, he chooses for any index i -- the probability that it can open in a way that verifies, in an accepting way, to both 0 and 1 -- so for every b , ρ_b is a valid opening, ρ_0 is valid for 0 , and ρ_1 is a valid opening for 1 -- that happens with negligible probability.

OK, so for every ρ , there exists negligible-- such that-- OK, so this is the collision resistance property. Any questions about the property?

OK, so now I have two things I want to do. The first thing-- oh, sorry, unordered, two things I want to do-- one is show you how you construct this thing, and two, show you how to use it to construct an interactive argument. Any preference on the order, what goes first?

Who wants construction first? Who wants interactive argument first? OK, we'll do interactive argument, and then we'll do construction.

It's not that-- if anybody wants to weigh in, say so, because it wasn't very significant, the result. OK, so interactive argument first? Yeah? OK. OK. So here is the-- this is what's known as the Kilian-Micali protocol.

OK. OK. So actually, I want succinct interactive arguments for NP. OK, so I'm going to show you how to construct a succinct interactive argument for any NP language. OK. So here is how I'm going to do it.

The prover, first-- so we have a prover and a verifier, and here is what they both share, a witness x . And the prover wants to prove to the verifier that x is in the language, and he has some witness. OK, and you can think about it as a 3-CNF-- can talk about 3-CNF and pick of it anyway. It can be nice to have something concrete in mind so you can think about x as being a 3-CNF and w as satisfying the assignment.

OK. What does the prover do, the first thing? Any guess what the prover first does?

AUDIENCE: Compute the PCP.

Yael Kalai: Thank you. Compute the PCP, that's the first thing he does. So he takes w and converts it to a PCP. OK. And if you want to have something concrete in mind, he opens up the GKR, adds a low-degree extension of the witness. That's a huge thing.

That's a PCP. He can't ship it over. That's too big. What does he want to do instead?

AUDIENCE: Hash it.

Yael Kalai: Hash. OK. He needs a hash key. He's like, I want a hash, Can anybody give me a hash key, right? Oh, let me-- sorry, because it's an argument, I should say they also share security parameter.

So the first thing the verifier does is it says, sure, I'm going to-- here's a hash key. I'm going to generate it, and here you go. Now what does the prover do?

AUDIENCE: Hash.

Yael Kalai: Hash. Thank you very much. So I compute. He sends over v . Usually, it's an-- OK, he sends v , which is Eval of hash key and the PCP, π .

OK. So the verifier gets this v , this digest. It's a bunch of bits with no meaning. He wants to open this PCP, right? He wants-- OK, so what does he do?

The verifier gets the v , doesn't understand a thing. What does he want from the prover? What will he ask the prover?

AUDIENCE: Randomly selects a bit.

AUDIENCE: Open [INAUDIBLE] to open it.

Yael Kalai: Exactly. He randomly selects a bit and has to open it. Which distribution? How will he choose which bits to open from this PCP?

AUDIENCE: [INAUDIBLE] whatever-- the PCP asks to call it oracle, right?

Yael Kalai: Exactly. What the-- exactly, as the PCP verifier, so what he does, he runs the V PCP. And the PCP verifier tells him, oh, sure, I want to know some ϵ .

So, OK, let me just mention, sorry, I'm going to construct an interactive argument for NP from two ingredients. Yeah, one, I'm going to use a PCP, so we have a PCP in our heads, the GKR PCP-- yeah, we have it-- and, two, this collision-resistant hash with local opening. These are two ingredients that we're going to use in this construction.

And we're going to assume-- remember, we talked about PCP when we defined it. It was before we went to kind of crypto land. And we said there's two parameters, c and s , completeness and soundness.

We said that a PCP comes with two parameters, completeness and soundness, c and s . And usually, we think of the completeness as 1, because all our PCP have actually completeness 1. It's easy to construct them with completeness 1.

Soundness, we pay for soundness. So the more queries you get, the better soundness you get. And here we're going to always assume that the soundness is negligible.

OK. So we give enough-- so for example, if we do one GKR, if you just run the GKR verifier once, if you remember, you kind of query one point and the low-degree extension. But a field element in GKR is of size $1/\text{polylog}$. So the soundness is still get-- sorry, the field size is polylog , so the soundness you'll get is $1/\text{polylog}$, which is not great.

I want negligible. So I'll need to repeat the GKR enough times, like, \log times, so that it will become negligible. I want my soundness to become negligible. The reason I want the soundness of the PCP to be negligible is because when we defined interactive arguments, we wanted soundness to be negligible.

OK, and these two will kind of-- the soundness I get from my interactive argument will kind of depend on the soundness I'm going to get from my PCP. So I'm going to assume here that I have a PCP with negligible soundness, that I ask-- so if I have a PCP with soundness half, I'll just repeat it polylog -- like, I'll repeat it, let's say, λ times, and now my soundness becomes $1/2^\lambda$.

OK. So I'm going to always assume that it's negligible in λ . OK, so I run the PCP verifier-- yeah?

AUDIENCE: So c minus the oracle access to the proof?

Yael Kalai: What do you mean? What is the previous version?

AUDIENCE: The previous version, where you had to query the proof oracle.

Yael Kalai: OK. So when you-- OK. So you said, let's remember the PCP model, where I had the PCP as an oracle, and I queried it. That was very succinct.

But if I had an oracle, in which world do I have an oracle? This oracle is a nice-- like, who puts the PCP in the sky for me? Where is the sky?

It's not sitting on a cloud. It's like a-- well, cloud, I guess, now has many meanings, so it can be sitting, actually, on a cloud. But that's kind of the point. We don't want anyone to store this thing. We don't want to assume someone actually holds this thing.

So in that model where you assume you have an oracle, everything is succinct. Now what I'm trying to do here is get rid of that assumption that I have an oracle. Great. Yeah?

AUDIENCE: What about even before the PCP, if you were to just do interactive GKR on the circuit [INAUDIBLE]?

Yael Kalai: OK. Good, good. Great, great. So the question is, look, just do an interactive GKR on the circuit-- on this c sub ϕ that may be here. Oh, no, it vanished.

You can, but here's the problem. You can do GKR, and essentially, it's what we're doing in the PCP, in some sense. But here's the issue.

But you're saying do you care. You don't need to deal with hashing, right. That's kind of your point-- a very, very good point. However, in GKR, anybody see-- that's actually a good-- anybody see what's wrong?

So let me repeat your point. The point is the following. Look, GKR is succinct. Think of 3-SAT. It's very low depth. It's succinct. The communication is very succinct. Why are you now going to crypto and opening GKR just to be done with it? What are we doing all of this?

AUDIENCE: You need the witness.

Yael Kalai: Very good. Why? Because GKR works if the verifier knows the witness, if the verifier knows the input.

In GKR, when we talked about GKR, we said, look, the verifier and prover both hold an input. And the verifier wants to know the output of this circuit and the input that he has.

And indeed, at the end, the verifier needs to check the final thing against his input. Is it, in the end, going to get kind of a value in the low-degree extension? And he checks, oh, is this consistent with the input I have in my hand?

Now he doesn't have an input. So what do you do? So, actually, let me tell you something.

The way these things are actually implemented-- it's a very interesting that you say that, because the way they're implemented is actually more along what you said. What they do is first they commit to just the witness. You're saying, why are you committing [INAUDIBLE] PCP? If you have a witness, commit to that, or commit to the low-degree extension of the witness.

And actually, in practice, that's what they do. They only commit to the low-degree and then they apply this interactive GKR. Or now they have even better protocols, IOPs, but that's for another time.

AUDIENCE: So you're saying that once you can commit to the witness and you do normal GKR, and then you only need to open the bits and the witness that you care about.

Yael Kalai: Exactly. So what you can do, you can-- first, the prover can tell the verifier-- sorry, the verifier tells the prover, you claim that you have a satisfying assignment. Give me a low-degree extension of your assignment, but hash it down because I don't want to hold this entire thing.

Now I have a hash. Now we're going to do GKR. Do GKR, then I need to check a random point in the-- a random point in the low-degree extension. I'll ask the prover, now open. But now you're committed ahead of time.

AUDIENCE: And then you have to make sure it was low degree.

Yael Kalai: You need to add a low-degree test to it. Yeah. Exactly. So, many of the implementations, that's exactly how they work.

And actually, a lot of the work currently, in the last couple of years, is actually not so much on improving GKR-- that, too, but a lot of the work is actually on the commitment, How do you construct this commitment that's kind of the most efficient? and so on. OK. Great. So-- yeah?

AUDIENCE: Do you not give the verifier the witness because the witness is too large?

Yael Kalai: Exactly. Exactly. We don't want to give the verifier the witness, because it's too large.

AUDIENCE: So we don't have zero-knowledge properties.

Yael Kalai: I don't care now about zero-knowledge. Usually, when people implement this, they add zero-knowledge on top, kind of as a-- but for now, I don't care at all about hiding. I just want verification, like, just to verify correctness. Yeah. No, like, all of the hiding zero-knowledge is kind of orthogonal. Yeah?

AUDIENCE: So the commitment that's given to the verifier is only used for verification. It's never actually opened. It's never opened completely. Yeah, it's never-- good point. So what you're saying-- this hash will never be completely opened.

OK. So the prover computes this PCP. He sends it over. The PCP will never be opened in its entirety. It's too big, or the witness-- again, never going to be open in its entirety.

Instead, what happens is the verifier generates PCP queries, and he sends the PCP queries i_1 up to i_l . And what the prover does, he only opens the locations π_{i_1} up to π_{i_l} . So he only gives him the PCP and these locations and opening, ρ_{i_1} up to ρ_{i_l} . Yeah?

AUDIENCE: This is a minor detail, but it was mentioned before that when you query everything in parallel, sometimes it's not sound. Can these be sent at the same time, or would they have to be sent through multiple, sequential rounds?

Yael Kalai: OK. So if you do a PCP, then they can be sent exactly the same time, because a PCP was committed to the PCP, and we're done. If you do GKR, you need to do everything sequentially.

AUDIENCE: Right. OK.

Yael Kalai: Yeah, if you did-- so if you did the version that was proposed here, where you don't do PCP at all, you have your witness, and then you need to do GKR completely interactive. But if you have a PCP, then it's committed. Now you don't have to do anything sequentially.

That's it. The guy is kind of stuck. So you can do everything at the same time.

So, great, so you send all the queries, you get back the answers. What does the verifier do. How does he decide if to accept or reject? What tests does the verifier do? Yeah?

AUDIENCE: Verify that the opening is correct, and then runs the PCP verifier.

Yael Kalai: Fantastic exactly. What does he do? He runs two things. He runs the PCP verifier with kind of i_1, i_l, π_{i_1} up to π_{i_l} , with x also, x comma, and checks that it's 1. And it also checks all the openings. So we check that $\text{Verify of hash key } v, ij, \pi_{ij}, \rho_{ij}, \text{ equals } 1 \text{ for every } j$.

AUDIENCE: Would that be π_{ix} ? [INAUDIBLE]

Yael Kalai: π_{ij} is kind of-- oh.

AUDIENCE: [INAUDIBLE] you add one should that be π_i ?

Yael Kalai: No, because when you verify a PCP, OK, the verifier never knows the entire π . He never reads the entire π . But to verify, of course, he needs to know the instance. You need to check it against the instance.

So the proof is succinct, but the verifier's work, he has to read what the claim is. OK. So the verifier will run in time linear in x . That, he needs, to read what the claim is. But the communication is very small.

So the communication is hash key, which is poly security parameter; v , which is a security parameter; i_1 up to i_l , which is going to be at most security parameter; and the opening, which is a security parameter. So everything here only grows with the security parameter, OK, polynomial security parameter, not with x . So the security parameter is polylog n -- n being the length of x -- then you'll have communication polylog n .

To verify-- well, OK, this verification is of size polylog n , like security parameter, because the x doesn't come in. But, of course, to verify the PCP, you need to read the input. You need to read the claim you're verifying. So here, you do run in time linear. These PCPs usually-- the runtime of the verifier is linear in the input length.

And let me mention, actually, there's this notion which we won't go into this semester, but it's called PCP of proximity, where the verifier doesn't even need to read the input. He just reads a little bit.

But now, of course, if he only reads a little bit, then how does he know that the input is correct? Maybe he didn't read one bit, and that flips it from true to false, you know? So the only guarantee he has that if he's accepted, then this input is close to an input in the language, that's why the proximity.

So there is a notion where the verifier is actually sublinear in x . He runs in time much less than x , but then the guarantee's weaker, the soundness guarantee's weaker. The soundness just says that if you accept, you cannot accept statements which are far from being in the language, but you may be able-- you may accept sentences-- you may accept claims that are false but close to ones that are in the language.

So it's a proximity-type soundness guarantee, but without-- if you really want soundness, yes or no, you have to read the input. There's nothing you can do, unless the input is kind of in an error-correcting code or you changed something about the model. But otherwise, you have to read the input.

Any questions about the construction? Forget about soundness and so on-- just about the-- like, about the proof, just about the construction. OK. So I just want to point to one thing that I said, let's send i_1 up to i_l .

Really, the way the PCP verifier works is he gets his input like-- let's say the security parameter just didn't know how many times to repeat the-- if the original PCP-- if the original PCP, let's say, is of size-- has soundness half, he needs to know how many times he needs to repeat it, so there's some security parameter. And some, he also knows the input length, like n in binary. So this can be very efficient, and he uses randomness.

What my point is, instead of sending i_1 one up to i_l , instead, he could have just sent the randomness. OK, he could have sent the randomness and told the prover, OK, you see this randomness? Compute-- run this yourself.

OK. Why am I saying this? It's like, OK, fine, why is that interesting? The reason this is very interesting is because-- and also here, by the way. Here, instead of the-- this is a randomized algorithm.

Actually, here, it's not important. Let's just leave this as is. I want to focus on this here.

Why is this important? Because later-- not today, but later in this class-- we'll start, actually, next week. We'll see that a-- we'll want to go-- after we'll prove this, and we'll be very happy, and we'll say, wow, we have a succinct argument and it's so succinct, we can have the security parameter-- success, happiness. But then, of course, every time we have something, we want some more, and then we'll say, you know what, we don't want interaction.

Why interaction? We want to go back to kind of actual proofs, so we want to eliminate interaction. And what we'll see is a way to eliminate interaction from protocols where the verifier's messages are completely random, OK, and that's why I want you to remember in your head that here, this message can be completely random.

OK. We can make also this random, but we don't need to, because it's the first message. It's kind of-- the first message is kind of special in that it's kind of the first one, so we don't need to eliminate interaction from it. It's kind of the first one, so that's fine.

All the later messages of the verifier will need to-- if they're really random, just random bits, then we can show how we can eliminate interaction, that we can actually get rid of the interaction. So I just want to point out, this message is completely random. These, i_1 to i_l , are not random. They're actually correlated. But you can just send the randomness. Yes?

AUDIENCE: I guess this wasn't assigned, but why is the first message special? Because you could imagine that, I don't know, there could exist some particular hash key which is really bad or something. So if you-- like, it feels like you should still be sampling the first hash [INAUDIBLE].

Yael Kalai: Oh, you're definitely sampling it. No, no. You're going to sample this key, for sure. I'm just saying, the hash key itself is not necessarily random bits.

It depends on the hash. Some hash keys are random. Some hash keys have some structure, depending on which hash function you use.

So some hash keys, indeed, some hash keys are just random, and then this message is just random. Some hash keys are not random. I have an LWE structure, for those who know what LWE is, but have some special structure. So now I'm saying-- I guess what I was saying is if the message is from the verifier to the prover, we're truly random, we're going to see we can eliminate interaction. So I'm telling you, just notice it's completely random.

And now you're going to say, wait, this is not necessarily random. I don't know how hash keys are distributed. Depends on the construction. Maybe it's not completely random. Then I tell you, don't worry, this doesn't have to be-- it can be an arbitrary distribution.

AUDIENCE: But, I guess, since the n is already random, why can't you just send the bit-- the coins you used to [INAUDIBLE].

Yael Kalai: Good, good, good, good, good. So you're saying, hey, send the coins here, too. Do the same. Put here r and send r .

You can. However, not all hash functions-- there are many hash functions that are secure if you only give the hash key, but I'm not going to give the randomness used to generate the hash key. Maybe with this randomness, you can break the collision resistance.

So some hash functions are OK. Some are not. Like, the hash function that we'll construct, you'll see, is fine. Actually, it's randomness. But that's under specific assumption. And some are not-- it's not OK to give the randomness.

But I just want to point out that for later, to eliminate interaction, using what's called the Fiat-Shamir paradigm-- we'll talk about it next class-- the first message actually doesn't need to be random. It's only the later messages have to be random. So I just want to emphasize this is random. There were other questions? Yeah?

AUDIENCE: Yeah. The PCP, the verifier, what is n ? What is the--

Yael Kalai: n , n is the size of x , yeah. So, sorry, I said, yeah, n , thank you, is the length of x . Thanks. So the PCP verifier needs to know the length of the instance he's working with to know which queries to ask. Yeah, Tina?

AUDIENCE: Why is the [INAUDIBLE] for the instance given in binary but the security parameters given in generation [INAUDIBLE]?

Yael Kalai: OK, because-- yeah, good. So I just want to emphasize that to generate these queries, you don't need to run in time that's polynomial in n . It's enough to run in time polylog n in order to generate the queries. That's why I gave it an-- it doesn't matter so much, because here, anyway, you need to run in time linear.

So you're like, why am I making this point? Yeah. It's not very important, even though sometimes it is important. When later you want to talk about proximity and stuff, then all of a sudden, it becomes an issue.

So, yeah, just, this is just emphasized that the PCP verifier, at least when generating the queries, doesn't run in time linear in n . It runs in time, actually, linear in security parameter and polylog in n to generate the queries. OK. Questions?

OK. So let's go ahead and prove the-- let me see. I changed the order, so I just want to make sure I'm not skipping. OK. Yeah. Let's do it.

OK, soundness, let's analyze this. So I want to prove that it's complete and sound, this protocol. Yeah, so if it's an NP language, your favorite one, let's say 3-SAT, and I want to argue that if the prover is honest, he indeed has a satisfying assignment or a witness, the verifier will accept him with probability 1. And if he's honest, he does everything like he should, and the soundness, if he tries to cheat and gives me an x not in the language, we're going to catch him. He's going to be accepted with probability negligible.

OK. So that's the goal. Any questions before we-- no? OK. So let's do it.

So, OK, let's start with completeness, because that's really easy. The focus-- well, I can just say, completeness is not worth writing, because it's so trivial. So what do I want? What does completeness say?

Completeness says, if the prover chooses x that's in the language, and he has a witness, he's going to be accepted with probability 1. Why? Why is he going to be accepted? Well, let's see what he does.

He chooses a PCP. The PCP, let's say, has completeness 1. That's my assumption. It's soundness negligible and completeness 1.

OK. I assume that about my PCP. Otherwise, the completeness is like the completeness of the PCP. So he chooses the PCP. He hashes it.

Now he gets PCP queries from the verifier. He opens the PCP. Now, these are acceptable with probability 1.

Now he generates also Open. These rhos are from Open. Right, so maybe I should write it. I didn't write it.

But ρ_{ij} is by computing Open and hash key π_{ij} . That's the-- yeah. That's how he generates ρ_{ij} .

And what do we know? We know, by completeness here, or correctness, he's accepted with probability 1. So the probability that both the PCP is accepted, the PCP verifier accepts, and the hash verifier accepts, is 1 because both of them is 1. I took perfect completeness for both of them, so that's all we do. So this is just pretty trivial.

Yeah? Questions? Are you guys OK moving to the soundness? That's kind of the interesting part. OK. So how about soundness?

OK, so what's the idea? The idea for soundness is the following. I want to say, so suppose there exists a cheating prover that chooses x not in the language, OK, and manages to cheat.

So let's suppose and let's try to break the collision resistance. OK. So suppose there exists t time whatever-- so, OK, we assume this is with T soundness, or security, and CR, collision resistance. That's my assumption.

Now, I want to argue, if I assume that the collision-resistant is T secure-- namely, even someone in time $\text{poly } T$ cannot cheat-- then I want to argue, suppose there exists a -- oh. Sorry. We're going to-- OK. We're going to get back to this, OK, what exactly this is, and we'll see that it actually needs to be quite big. But we'll see that.

Suppose there exists a $\text{poly-}T$ cheating prover P^* , that cheats, namely that-- and there exists, like, for every λ -- or there exists a bunch of x_λ not in the language. And there exists a non-negligible ϵ such that the prover succeeds with that probability. The probability that P^* talking to verifier on input x , the verifier accepts him, is at least ϵ , such that for every λ -- to be precise, this is x of λ . And they get λ .

OK, so suppose a cheating prover, for any λ , he chooses some x of λ and P^* , and V accepts him, even though x_λ is not in the language. OK. One, my only assumption is that x_λ is smaller than 2^{λ} . So I choose my security parameter so that the [INAUDIBLE] number is bigger than \log the size of x .

OK. Then-- actually, it's not even needed for soundness. This actually was needed for completeness, because we only had completeness for these x 's. For soundness, actually, it doesn't really matter.

OK. So let's see. What do I want to do? I'm saying suppose there's-- because this is a cheating prover, and this cheating prover convinces.

OK. Now, let's see, what did this cheating prover do? He committed to a P kind of-- he hashed a PCP. Now, I don't know what he did. He's a cheater. But he gave me a hash of something, should be a PCP.

And then he opens this PCP. So what do I want to do? I'm going to-- I want to argue I can find a collision.

Why can I find a collision? What is the idea? The idea is the following.

I'm going to run this cheating prover. I'm going to ask the cheating prover, OK, here's a hash key, give me a v . Then I'm going to run him many, many, many times on many PCP verifiers.

I'm going to tell him, OK, you know what, here's i_1 to i_l from the PCP verifier. Give me openings. Again, here's i_1 up to i_l . Give me openings.

Again, give him the-- I'm going to kind of rewind him and try to give him tons of kind of messages, i_1 up to i_l . So I'm going to kind of rewind him. OK. So again, what do I want to do?

Given a hash key, OK, given a hash key, I want to find collisions for this hash key. How do I find collisions for this hash key? I'm going to give this prover the hash key. He gives me a value v .

Then I'm going to tell the prover, I'm going to choose a random r . I'm going to tell him, open these locations. He opens.

Good. Now I'm saying, OK, OK, OK. Actually, let's rewind him. Forget that I sent him these. Let me give him r_2 , different r 's-- Open. OK.

Forget-- r_3 -- I'm going to run this step tons of times-- r_1 , r_2 , r_3 , r_4 , ba-ba, ba-ba, a lot of them. And I'm going to have a lot of openings from him. I'm going to run it so much that I'm just going to have tons of opening. Like, I'm going to run in more than, like, $\text{poly } 1/\epsilon$ and more than $\text{poly } 1/\epsilon$ in n . So the point is I'm going to run it much more than-- like, the size of the PCP times $1/\epsilon$, because it's some power to some-- to the third, whatever. that will take a lot of time.

Now, what's the idea? The idea is the following. If there are no collisions, so if there-- look, I'm going to run.

Now, if at any point, every time that he gave me an opening that didn't work-- it's 0, it didn't work-- OK, so I'm going to throw it out. OK, look, there's-- he only succeeds in probability ϵ . So all the times that he didn't succeed-- OK, well, failure. But $1/\epsilon$ is the time he succeeded.

And in all these times, in all these $1/\epsilon$ -- sorry, ϵ is the time he succeeded. In all these ϵ times, I got a lot of good openings. Now let's look at all these openings that I got.

If somewhere there's a collision, I found him. I found a collision, so I'm happy. If nowhere there's a collision, then essentially I found a PCP from him.

I completely constructed, like, maybe not the entire PCP, but a lot of it. And the rest, I'm going to put zeros. I say, you know what, OK, these are identically zeros. But the argument is that if there were no collision, I get a PCP that's accepted with probability at least $\text{poly } \epsilon$, which is non-negligible.

So that's the high-level idea, but let me say everything I said now, slowly. OK. So suppose I have a cheating prover that succeeds in convincing for x not in the language. I'm going to construct-- I'm going to find a collision to the hash function.

So we're going to construct adversary A that breaks the collision resistance property. OK, how do I break the collision resistance property? Here's the idea.

So, A , OK, so given, so he's given a random hash key, and his goal is to find collisions. How does he find collisions? What does he do?

He gives-- so here's what A does. First compute v , which is P^* and hash key. That's the first thing.

I've got a v . Then I do the following, many times. So for i goes from 1 up to $\text{poly}(n)$ and $1/\epsilon$ -- so, many times-- what I do is I run, like, What P^* , given hk -- and he gave v -- would run on input r_i , r_i corresponding to-- OK, let me call this j -- j corresponding to kind of i_1 up to i_l . I just didn't want confuse this i with these i 's.

OK, so I'm going to run many times. I'm going to choose many r 's. Oh, I didn't say-- sorry. Sorry, for i goes to 1, choose a random r_j . So for j , go from 1, 2, up to $\text{poly}(n)$ and $1/\epsilon$. I choose a random r_j .

How much random bits? Like the PCP verifier, I'm going to behave like the-- I'm going to give him the randomness of the PCP verifier corresponding to i_1 up to i_l , and I'm going to compute the his answers, which is π_{i_1} up to π_{i_l} .

These may be malicious answers. I denoted by π_i , but of course, it can-- I mean, it's not necessarily corresponding to a nice PCP. He's malicious and opening.

OK, so this is how I find collisions. I get a hash key. I give it to the cheating prover to get a value. Then for m times, for j goes from 1 to m -- m is going to be $\text{poly}(n)$ and $1/\epsilon$ -- what do I do?

I choose randomness for the PCP verifier, and I tell the prover, What are your answers for this randomness, with the same v ? I didn't change the v . OK. So he gave me a v in the second message, and then I kind of run the third message with him many, many, many times, n times, and expect to see answers.

OK. Now what do I do? Those for which the ρ doesn't work, those for which ρ doesn't accept, it has a bad-- the Verify doesn't-- this doesn't accept, I throw out, then it's meaningless. He could put whatever he wants. I throw it out.

But that's only ϵ fraction of them, because it turns out the prover, I assume the prover is accepted with probability ϵ . So ϵ , I throw out. OK, but I chose much more than $1/\epsilon$ -- OK, this only affects-- you know. So except for these ϵ , the rest are good. Yeah? Sorry.

AUDIENCE: [INAUDIBLE] threw out $1 - \epsilon$.

Yael Kalai: I throw out--

AUDIENCE: [INAUDIBLE].

Yael Kalai: As I said that, I-- yeah, I throw out-- I'm left with ϵ . OK, I throw out $1 - \epsilon$. I'm left with ϵ .

But epsilon is good enough, because I chose more than $1/\epsilon$. That's why I have here $1/\epsilon$, because epsilon are bad, which is why.

OK, so I still have a lot left. OK, so now what do I do with the ones I have left? So I want to argue-- so here's the claim.

The claim is that-- let me write it here. So the claim is that I should be able to find collisions. So the algorithm, the algorithm A, what it does, the way it breaks-- and so find output, a collision, if one exists-- I'll write it here. Why am I-- output, so 3, output collision if one exists.

That's it. So what do I mean, if one exists? Many of the i 's here-- I asked many times, right, because I did poly in n -- n meaning kind of the witness length, yeah, more than-- think of the witness length to the power of $1/3$ divided by epsilon. The size of the PCP divided by epsilon to the third, that's how you should think about it, or times, like, security parameter.

OK, so now what's the point? The point is I saw the same i many, many times because I asked many, many queries. Now, I want to argue that if-- so there's two options. Either every time I ask for an i , if he gave me a valid opening, it's for the same one, or one of the i 's, he gave me valid openings, one for 0 and one for 1.

If that's the case, I won. I found a collision. And then I broke the hash function. So if you believe you cannot break a hash function, then you cannot break. Then the scheme is secure, which is kind of what we're trying to do.

But now you can say, but maybe you didn't find a hash-- maybe you didn't find a collision. Maybe the cheating prover, whenever he answers, when he answers correctly, it's always with the same-- like, for every i , if he answers correctly, it's with the same π_i . He never answers correctly with π_i being 0 or π_i being 1.

Then what happened? I take all the things that he asked me, and I pieced together a PCP from him. OK, so why?

I say-- OK, so the question is-- suppose I output-- suppose that a collision happens with negligible probability. If it's non-negligible, I won. I'm done. OK. I want to argue that now, if not, if a collision-- so if a collision occurs--

AUDIENCE: Two Cs.

Yael Kalai: Thank you-- with negligible probability, then I want to argue I can come up with a PCP, then there exists a π that I can construct from this adversary such that V PCP of π accepts with non-negligible probability. And that's a contradiction, because I assumed that my π is sound, with-- I mean, soundness holds with probability. You break soundness only with negligible probability.

So I'm going to argue that if the probability of finding a collision is negligible, then, actually, I can come up with a PCP that's accepted with non-negligible probability. But x is not in the language, so that breaks my-- that can't happen. Because I know the PCP has soundness, negligible, so that can't happen. And therefore, I must have collisions with non-negligible probability, and hence broke the collision.

So why is it the case that I can find a PCP that's accepted? Because what am I going to do? Remember-- let's think I never found collision. It's negligible, so for all practical purposes, it's never.

OK, let's think of the fact that we never encountered a collision. So then what do we do? I take this PCP. I take this adversary. I run it.

Oh, I start piecing up a PCP, and I say, oh, π_1 here, good, π_i . Every time I ask it again, I know it'll give me this answer, because there's no collisions. And I start slowly piecing together an entire PCP.

Now, there may be some places of the PCP that he will never open correctly, never. He'll never give me a valid opening. So then I can't-- I don't know.

I don't know. Moreover, there may be some places of the PCP that somehow this verifier never asked. That can also be.

So I'm going to write-- all the places that actually opened, I'm going to write down whatever didn't open. Look, this adversary can always-- let's say it can always take $1 - \epsilon$ fraction of this PCP and never open it, and the rest open correctly, because it's only a step of move for the ϵ . So there can be always-- not really $1 - \epsilon$ of the PCP because we asked too many queries, but there can, of course, be some parts of the PCP that he just never answers.

And it's OK because of this ϵ . OK. So in that part, I'm going to-- wherever I don't want, I'm going to just put zeros. OK, but this is my PCP.

And I said, now, this is fine. I'm going to call this π . Now you're asking, Why would the verifier accept π with non-negligible probability? Well, because this P^* is accepted with probability ϵ .

And, well, this-- V is going to ask-- OK, so what's the concern? The concern is, well, V now is going to ask, like, places that are 0 here. Right, so the point is we're saying-- your concern is-- look, P^* succeeded.

Maybe P^* succeeded. But here, if P^* succeeded, we have here our π 's, then P^* will succeed, too, because P^* answers like this. But you're saying, well, but maybe here there are zeros. Maybe here there are zeros.

But why are there zeros? Let's remember, why did I put zeros here? Because P^* failed on them.

That's why there are zeros there. I asked this place many, many times, and every time, P^* said no. So he won't succeed on this.

Or there's also another option, that V PCP never asks that question, in which case he still won't ask that question. So again, the claim is that this PCP will be accepted with probability close to ϵ , because the probability that we will hit locations where we fail and he succeeds is very, very, very small. So whenever we fail, he will fail, too, with very high probability, because if he succeeds there, we would have succeeded. We asked so many questions, we would finally get the right answer.

That's the high-level idea. OK. I don't want to go into more details. There's a link, in the website, exactly on the proof.

But there's two reasons I don't want to go into more detail-- first, because it's tedious, so if you're interested, just look at the paper. But also, this is not the right proof. This is not the right proof.

And why is it not the right proof? Because, note, I cheated you, actually. I actually lied.

I promised you that I broke soundness, but I didn't actually. There's a very minor-- I don't know if you'd call it "bug," but I lied. And this doesn't quite do-- it doesn't quite break the collision resistance with my-- OK, when I say A breaks the collision resistance, I need to say A breaks-- poly-T-lambda adversary that breaks-- right, I said, I assume that you cannot break the collisions in time T lambda.

So I said let's just say there's a T lambda times P star, and I'm going to break the-- I'm going to find the collision in time poly-T lambda. That's what gives me a contradiction. Right, I argue, my assumption is that there exists collisions in hash that cannot be broken, collisions cannot be found in time T. To break that, I need to say, if there is a cheating prover, then I can break the collision resistance in time poly T.

Now, let's look at the runtime of this adversary. The P times of time poly T. That's OK. But what does my adversary do?

My adversary, he tries to generate an entire PCP. And then he says, if I succeeded, I can succeed, and therefore, I must be collision. But to generate an entire PCP, he needs to run in time poly in n. What if T of lambda is smaller than n?

Now, you can say, eh, come on, T of lambda must be bigger than an n. Otherwise, this kind of-- there's different-- I don't like this definition, because we shouldn't allow the cheating prover to run in time poly n. If you don't, something's wrong with the definition.

I don't know, maybe yes, maybe no. I have no definition that says time T of lambda. Maybe you want T of lambda be smaller than n because the application makes sense, because you use it in a way that actually the prover will never run in time n.

So I have now some T of lambda. How it connects to N, I actually don't know. And I want to ensure that I can break the adversary-- I can find collision time T to the lambda.

But actually, I run piece time which is in time T to the lambda. That's fine. But I run it n times. $1/\epsilon$ is fine, because epsilon is like-- epsilon in lambda, or T to the lambda-- OK, it's $1/\text{poly } T$ to the lambda. Fine. That's good, T to the lambda-- T of lambda, sorry.

But this n, n can be bigger than T of lambda. So really, this works if-- this definition is good if, only if T of lambda is bigger than n. So this is a good proof if-- and this has kind of-- it's kind of bothered cryptographers, because even though it makes sense to assume that T of lambda is bigger than n, it makes sense to let the cheating prover run in time n, it's annoying that we need to do that.

So I think we should take a break, a five-minute break. After the break, I'll tell you just a very high-level idea. There's a beautiful paper that came in 2002 by Barak and Goldreich, that they showed how to get around it, how to do it for any t.

They get around-- they actually don't construct an entire PCP. They do what I think is more clever and more beautiful, kind of what I think of as the right proof for this. And we'll see that after the break, and then we'll do the construction of the collision-resistant. But there was questions before. Yeah?

AUDIENCE: Is the rewinding necessary so you get lower success probability but still polynomial [INAUDIBLE]?

Yael Kalai: Great question. The rewinding for-- OK, so the question is, Is the rewinding necessary? And it's a great question. The answer is, for all I know, yes, the rewinding is necessary.

This question was very important also for post-quantum, because today, we're worried about quantum computers. And if the cheating prover is a quantum device, you can't rewind him, because once you measure a quantum state, it collapses. Eh. And so proving post-quantum security of this protocol, it was-- actually, it's known to be post-quantum secure.

But it was a lot of work because it was work that required to kind of rewind the cheating prover in the quantum-- to rewind the quantum cheating prover, and, Why can you rewind? So they showed you can actually do it. But just to answer your question, Do you need to rewind? yes, kind of we don't have a straight-line proof of soundness for this protocol.

Audience: And is it because the witness [INAUDIBLE]?

Yael Kalai: You know, so it's-- the other proof I'll show you after the break doesn't quite construct the PCP. But in order to find collisions, we need to run them at least twice. I run them and then-- because how am I going to find collision?

I give him queries. He's going to give me answers. How can I find collision from that?

But if I run him once, and I run him again with-- the idea is I'm going to run him-- I'm going to choose, like, a random query i . I'm once going to run him i with some queries, and then I'm going to run him i with other queries.

And the only way you can cheat is if you're not always consistent. But at least, I need to remind you twice. So that's the-- any other questions? Yeah?

Audience: Could the right group [INAUDIBLE] query if we had a concrete hash function instead of this abstract notion?

Yael Kalai: No, I don't think the-- actually, when you look inside the hash function, it just becomes messy. I think actually abstracting out the hash functions, and just thinking what property it gives you, makes the proof cleaner. Kind of to put boxes around things and just modularize kind of what you need, it actually helps the understanding and simplifies the proof.

Yeah, I think the problem with this proof is that it requires constructing an entire PCP, and sometimes you're saying, Why do I need to construct an entire PCP? So the other proof kind of shows that you actually don't need to do that. And that's why the prover doesn't need to run time linear in n . It just runs in time that depends on t . Any other questions before the break? Yeah?

Audience: Just to clarify, n here is the size of the PCP, not the [INAUDIBLE].

Yael Kalai: Yeah, you're right. n is the size of the PCP. But I'm thinking here, because I'm thinking of 3-SAT, I'm thinking the input, the instance, and the witness and the PCP are all poly-related. So because I have here a polynomial, it doesn't matter.

But I do want to say, in some settings, you can think of the witness can be much, much smaller than the instance or much, much bigger than the instance. So for example, take nondeterministic time T . x is of size n , but the witness is of size T of n , which can be superpolynomial in n . Then this becomes polynomial in the witness.

The PCP is always-- or, OK, the PCP is always polynomially related to the instance and witness. Now, sometimes the witness is bigger. Sometimes the instance is bigger.

And the relation does not need to be polynomially related. If they're all polynomially related, it doesn't matter here, because I put a poly. But sometimes the witness can be much, much smaller, like a log clique. The witness is size log squared but the input is n . And sometimes the instance can be small and the witness can be big.

So what I mean here is input plus witness length. Here, I just put one of them because I assume, 3-SAT, they're both the same. So, yeah, but that's a good-- what really I need to put here is PCP length. That's the truth.

Yeah. Great. Great point. Fantastic. Are there questions before we break? OK, let's do a five-minute break, and then we'll return.