[SQUEAKING]

[RUSTLING]

[CLICKING]

RACHEL ZHANG: OK. So what is GKR? Can you guys read this? Is this too big? Too small? It's fine?

AUDIENCE: It's fine.

RACHEL ZHANG: OK. So the idea is given a circuit C, we're going to-- instead of hash computing circuit C by yourself, you can delegate it to a prover who would do the computation for you and prove the computation is correct in an interactive protocol. And the main idea of how you can do it--

So let's say this is the depth D and size S. What you're going to do is you're going to take every single layer of the circuit, extend it to a low-degree extension. So is this going to be the hat.

So you're basically going to take all the wire values on each layer, find a low-degree extension of it, and then you're going to have some low-degree extension of each layer. The top layer is going to be the V1, then V1 layer, and so on. The final layer is going to be VD.

And the way GKR works is essentially as follows. You're going to reduce some claims about the top layer-- the upper layer-- to claims about layers below it. And you're going to keep doing this until you reach the bottom layer, at which point the verifier can check themselves. So at the beginning, you might have that the output gate-- you want this to be equal to 0. You're going to reduce this, via sum check protocol, to two claims.

So let's call this V11 equals, maybe, the value of V11. And also, the claim that the low-degree extension on the second point-- so that's also equal to the different value of you want it here. And this is [INAUDIBLE] assumption protocol. And then you can do it again. So last time, [INAUDIBLE] two-to-two fashion. So we're going to reduce the two claims to two claims, [INAUDIBLE].

I'm just going to keep going. At the end, you're going to reduce the assumption to the claim about the output layer. So maybe ZD1.

And at this point in time, this is just a low-degree extension of all the input gates, which is size n. The verifier knows what all these inputs are, so they can compute the low-degree extension by themselves. And so all together, this is going to take a prover time poly (S) to prove the computation's done correctly to the verifier. And the verifier will run, in time-- let's see it's going to be D times polylog S, plus n times maybe polylog S, also. So this is what we did last time. Yeah.

AUDIENCE: Can you go over how using the same randomness help you get the thing-- you just do all the way down?

RACHEL ZHANG: Right. So the idea is, instead of having to run a different subject protocol with independent randomness for each of these two claims, what you can do is use the same randomness for both claims simultaneously. And turns out that the gates you have to check at the very bottom, at the end of sum check protocol, will be based on the random method that was used during from the verifier's point of view. So as long as the same randomness using both of these things, then you'll have to use the same two final checks at the very bottom of the sum check. Yeah. OK. Cool.

And maybe I should also just tell you-- yeah, I also wanted to write out sum check equation just so we have it for later because it's going to be important. Used up an entire board range is kind of awkward. So maybe I'll put it somewhere else. Yeah. Oh, it can move there. Maybe? OK.

I guess-- I told the camera man we wouldn't need that board. But I'll just put it here because it shouldn't be important, I guess. Or it would be important for us, but we did last time. So hopefully, it won't be new.

So you're checking an element of the ith layer. You want to reduce it to 2 and things in the previous layer. And so what you can do is you can take an output gate and two input gates, and it's going to be this giant thing. So whether or not it was at gate times-- times this one-- times-- whether-- times the sum of it.

And then you need to take a low-degree extension. And so that's the equation. What's going to happen is you're going to essentially do a sum check with this entire thing here as a polynomial in your sum check. So we'll come back to this later and we'll just leave it up for now.

AUDIENCE: Sorry. What's the Q2?

RACHEL ZHANG: Yeah, it's a polynomial of degree-- I'm going to say it's, like, the size of H, maybe, in each variable. So individual degree size h minus 1. Or just take it over-- to the M. And it's going to basically test the quality of z and p. So if z-- if p is in this hypercube, then it should be 1, if and only if p is equal to z, and then the low-degree extension of that.

OK, so today we're going to ask some questions. So we knew how to do this. So last time, I showed how to do this, as long as we had Oracle access to the add and mult gates. So the goal of the first part of today's class is to remove this Oracle access. We're going to show how to get these things. Actually, we have a little bit more in this board.

So I guess-- so question one is how to get these low-degree extensions. Maybe you guys can tell me. Do you have any ideas? How can we get these? So I feel like Arnan is bouncing, which clearly means he has something to say.

AUDIENCE: Oh.

[LAUGHTER]

Could you arithmetize the circuit's [INAUDIBLE] succinct description?

RACHEL ZHANG: OK, good. So maybe I'm going to say something like maybe you can compute these things yourself somehow, if you can arithmetize it. And we're going to see how do this in a sec. Another way to do it is we already know how to do GKR. Like, we already know how to delegate a circuit. So let's redelegate the add and mult gates.

So we're going to redelegate add and mult. And this is, indeed, what they do in the original GKR paper. But we're not going to do it today. So this is another thing you could do. But today, we're going to show how to compute them.

Actually, this is a bit weird because we're actually going to compute the low-degree extensions of these add and mult circuits. We're actually going to compute a different, somewhat low-degree polynomial on these-- that share the same values on the hypercube. So it's actually not going to be the canonical LDE.

I think last time, Leo had said something about-- can you use a different low-degree polynomial? And the answer here is, yes, and we're going to see application of that today.

OK, cool. So the idea is-- so if you want to compute them-- so you can't compute the actual canonical low-degree extension. So what can you do? Well-- and the reason is basically because if you compute the add and mult gates-- because the circuit is too complicated for you to compute yourself.

So maybe here's an idea. What if, instead of computing-- doing GKR in the original circuit, we do it on a much simpler circuit that computes the same functionality as a circuit you're trying to-- that you want?

So idea-- replace C with a simpler to describe circuit. And I guess by simpler to describe, I just mean that add and mult should be easily computable.

And so let's use the universal circuit. So universal circuit is going to be a circuit that's going to take as inputs the circuit that you're trying to compute, along with the input. And it's going to output-- it's going to simulate the computation of your circuit C on the input x and then compute C of x.

And the claim is going to be that this universal circuit is going to have a simpler description, at least for the add and mult gates, than the original circuit because it's going to take us input like the description of the circuit. And then it's going to simulate description, but all these simulation is easy to describe because it's very general and very easy.

So maybe let's see how this will work. So you have this universal circuit here. It's going to look like this. And it's going to take as input a description of the circuit C, along with your input x. And then it's going to run-- the hope is that this is going to be D times poly log-- maybe it should be-- OK. We'll talk about this later. So this should be, hopefully, not so much bigger than the circuit C because you're going to want to do it efficiently.

What is this description C over here? This is-- yeah, maybe I should-- OK. So what does the circuit C? What should input to this be like? Well, it's basically going to be a function that will tell you-- yeah, so I guess it would tell you-- for every single-- maybe this should be an add 1 and then mult 1.

So for both add and mult, it'll tell if, at a single location in the original circuit, C-- if there is a gate of that form or not. So it's going to be a pretty large description. It's going to be size 2 to the s. Does that make sense? Yeah.

AUDIENCE: So the circuit [INAUDIBLE] size. We're going to give a description of size 2 to the s.

RACHEL ZHANG: Yes. We'll start with that. And as Ted is alluding to-- so would not be so good for us, but we'll discuss how to do in a little bit.

AUDIENCE: S this is just--

RACHEL ZHANG: Yeah.

AUDIENCE: Just-- literally, just two-bit strings.

RACHEL ZHANG: Yeah, yeah, yeah. It's supposed to be really long. It's going to tell you what every gate of the circuit looks like.

AUDIENCE: Didn't pw1 and w2 range over the horizontal distance of the circuit, not the vertical distance?

RACHEL ZHANG: Yeah, OK. Maybe we can have pw1 and w2 range over all the possible gates or something. So we can change the notation a little bit. That's OK. OK.

AUDIENCE: Sorry. Last question.

RACHEL ZHANG: Yeah.

AUDIENCE: I guess this is a very small detail, but how do you handle knots at the input layer?

RACHEL ZHANG: So what you can do-- this is also, actually, a huge issue with the GKR protocol, because you only have add and mult gates. But you can do is you can have , and then also the flip of x or the not of x. And then now you can simulate everything.

AUDIENCE: OK.

RACHEL ZHANG: OK. So my claim is basically that this universal circuit is going to be much simpler to describe than the original circuit, C. And maybe let's just see why in, hopefully, a not too in detail view, but hopefully, we can get an idea. So the idea is-- let's say this is your circuit C over here. It's going to be a bunch of layers with a bunch of gates in each of the layers, and they're going to each have some functionality.

And so what the universal circuit is basically going to do is it's going to say-- I want you to compute this value of this circuit C gate here. Let me look at all the previous ones, because you don't know which one is a tattoo. So you're going to look at all the possible values, and you're going to, then, see what their circuit C says and then compute the appropriate function of the previous layer based on what the circuit C says.

So in particular-- yeah. So the particular in the ith layer-- so the value of the circuit-- of this thing in the ith layer-- is going to be-- so maybe this is a layer, and this is a gate. And it's going to be a sum of possible gates in the previous layer of the following thing.

So it's going to be whether C-- so it's going to be whether C has an add gate corresponding to the three gates you're picking. So this is J, And then maybe K and K prime, and then times the sum of them. So it's a similar thing as in the GKR thing. We're going to have an add and a mult condition.

So if it's an add gate, you're going to compute the sum. And if it's a mult gate, you're going to compute the mult. Is that OK?

AUDIENCE: What is that inside the parentheses on the add gate.

RACHEL ZHANG: Yeah, OK. So this is going to be whether the circuit is an add gate at these particular three gates-- J, K and K prime. And this is going to be the values, V I plus 1K. So it's what this value is.

AUDIENCE: [INAUDIBLE]

RACHEL ZHANG: Yeah, sorry So maybe this is a little bit small. OK. I don't really know how to make this bigger. Yeah.

AUDIENCE: The subscript on the C add-- I actually think of it as an input [? text. ?]

RACHEL ZHANG: Yeah, exactly. So these are both going to be given from the description of the circuit C, which-- this will also have to take-- the description of circuit C, and it's also going to read into here.

AUDIENCE: And this is the same thing that's written on the board [INAUDIBLE].

RACHEL ZHANG: Yeah, it's essentially the same thing.

AUDIENCE: It's just not [INAUDIBLE] section.

[INTERPOSING VOICES]

RACHEL ZHANG: Yeah.

AUDIENCE: Sorry.

RACHEL ZHANG: No.

AUDIENCE: I'm still hung up on this. C add is defined over three inputs, all of which range from 1 to [? test, ?] right?

RACHEL ZHANG: Sure. OK. Yeah.

AUDIENCE: Why is it a size 3? So it's essentially the same [INAUDIBLE] size like s cubed.

RACHEL ZHANG: Good. Yes. Yes. So we're going to have to deal with that in a bit. So yeah, instead of saying that the circuit C here is-- the description of circuit C is too big-- we're going to have to do something for that.

AUDIENCE: Wait. So why is it 2 the s, though? Is it because we have--

RACHEL ZHANG: Sorry. It's just s. It's just S.

AUDIENCE: OK.

RACHEL ZHANG: You're right. You're right. It's just S.

AUDIENCE: Wait. Wasn't it S-cubed?

RACHEL ZHANG: S-cubed? Oh, yes.

[LAUGHTER]

Does that makes sense? Sorry. I tried to explain that [INAUDIBLE] really badly. But you're basically trying to simulate the computation of C. And to do that you're going to introduce a new circuit-- like, a new small circuit for every single gate in your universal circuit that will look at the previous layer and then compute appropriate functionality based on [INAUDIBLE] is supposed to say. OK.

AUDIENCE: Rachel?

RACHEL ZHANG: Yeah.

AUDIENCE: What is the p [INAUDIBLE] on this?

RACHEL ZHANG: Yeah, OK.

AUDIENCE: On the off-camera board.

RACHEL ZHANG: Yeah, yeah. So-- yeah-- so these are all going to be H to the m. H to the m here is going to be the size of, like, a layer.

AUDIENCE: OK, OK.

RACHEL ZHANG: So it's going to be-- so p you can think of as being in the ith layer. And then omega 1 and omega 2 are in the i plus 1th layer.

AUDIENCE: I see. I see. And p to z is just an arbitrary [INAUDIBLE].

RACHEL ZHANG: Yeah.

AUDIENCE: OK.

RACHEL ZHANG: Yeah.

AUDIENCE: So that summation there really represents-- you're trying to compute that summation with a bunch of gates.

RACHEL ZHANG: Yeah, yeah, yeah. So this is going-- this is going to correspond to-- I'm sorry. This is really bad. Yeah, this is going to correspond to a pretty big circuit here that's going to look at the previous layer, and-- yeah.

AUDIENCE: So the depth of U could be, like, the depth of that mini circuit times p.

RACHEL ZHANG: Exactly. Yeah, that's perfect. So-- yeah. So I don't know what your name is.

AUDIENCE: Gabe.

RACHEL ZHANG: Gabe. So as Gabe said, the depth of the circuit is going to be-- the depth of the original circuit of C times this depth here. So the question is, what is the depth here? Thoughts? Roughly. I don't know. [INAUDIBLE].

AUDIENCE: log S?

RACHEL ZHANG: Yeah OK. Something has log S. Like, polylog, maybe, at worse. Yeah, and the reason is because you can think of this big sum as a binary tree of sums. And then at the very bottom of this binary tree, you have poly S many things you're looking at. So this entire depth, because it's a binary tree, is only going to be to depth log S or something.

AUDIENCE: What's U?

RACHEL ZHANG: The depths of the universal circuit.

AUDIENCE: OK. Good. U.

RACHEL ZHANG: Yeah. And what about the size of this universal circuit?

AUDIENCE: [INAUDIBLE]

RACHEL ZHANG: Yeah, so it's going to be the size of original circuit times, like, poly S, so it's just going to be poly S. And the reason for that is because the size of this universal circuit is basically going to be-- for every gate of the original circuit, you have-- this poly S-size circuit's replacing it. So you get to poly S size here and then times the size of the circuit, which is poly S.

OK. So this is going to be like part of the circuit we're going to end up using to delegate, instead of using the original circuit, C. I guess the last claim that I made is that this circuit U is much simpler to describe than the circuit C. So particularly, the add and mult gates are easily computable or describable, very simply.

And maybe let's look at this and think about it for a minute. So I guess if you're trying to figure out-- I have-- I'm trying to look at this specific gate, and I want to see if it's, like, an add or mult gate. But you do, and you figure out where in this tree you are. And you can be like, oh, look, I'm part of this binary tree. It should be an add gate. And then you're like, OK, I'm an add gate.

Or you can read, maybe, in the leaves. You're like, oh, this particular gate corresponds to some add gate in the original circuit times some values. And you figure out-- oh, this is supposed to be multiplied or add gate or something. The point is that this is a pretty easy function to describe. In particular-- so there's going to be low-depth Boolean formulas that compute both the add and mult gates.

So there's going to be a Boolean formula of-- let's see. So it's going to be a Boolean formula of size polylog S that computes the add and mult functionalities.

So it tells you that-- within U, these-- because they're small formulas that tell you whether a particular gate in U is an add or mult gate. That good? Cool And so now, like these formulas are much easier to compute than the previous add and mult. So ostensibly, the verifier can do it themselves now. So--

AUDIENCE: [INAUDIBLE]

RACHEL ZHANG: Yeah.

AUDIENCE: Can you use [INAUDIBLE]?

RACHEL ZHANG: OK. So I guess, originally-- yeah, originally-- so how do you compute add and mult? We're going to-- I didn't write it yet, but you're going to assume that z is a logspace-uniform circuit, which means that you-- you can do maybe up to time poly S computation to compute it, which is too much.

Here, since the circuit-- the formula is only of size polylog, you can compute it yourself. So then when you're doing GKR in the universal circuit, you just have to do a small computation as opposed to the big one. Yeah.

AUDIENCE: Are these-- if we add [INAUDIBLE] mults it's a logspace uniform?

RACHEL ZHANG: I guess maybe a log logspace uniform, right? Is that how it works? [LAUGHS] I mean, there's is polylog, right? So basically, what I mean by this is you can feed in any p omega 1 and omega 2 and then compute a formula of this size and it'll tell you what the answer is. So it's only polylog size. So you only have to--

AUDIENCE: Does there exist a formula such that, for all p1, omega 1, omega 2--

RACHEL ZHANG: Yeah.

AUDIENCE: --the formula computes it, correctly?

RACHEL ZHANG: Yeah, yeah.

AUDIENCE: Right. But-- yeah. The naive way I would do it is advice of polylog size. But I'm assuming there's a way to make it-- to get rid of this advice, right? [INAUDIBLE]

RACHEL ZHANG: Yeah, I think it's log, log uniform.

AUDIENCE: I think it's just, like, [? max ?] If-- to figure out what level i is to figure out what [INAUDIBLE] is.

AUDIENCE: [INAUDIBLE] universal circuit.

AUDIENCE: Yeah, that's right. Yeah, that is.

AUDIENCE: Oh, the size for the universal circuit.

RACHEL ZHANG: Yeah.

AUDIENCE: OK. OK, I understand.

RACHEL ZHANG: Yeah, maybe-- just to summarize what we've done, we've essentially replaced this C, which has complicated add and mult gate description, with a universal circuit, which has a much simpler add and mult gate description. And then, now, what we're going to do is we're going to run GKR on the universal circuit instead of the original circuit, C. The output is still the same because the universal circuit will simulate C and compute C of x. But now, the verifier can, themselves, compute. add and mult.

AUDIENCE: But the input [? N ?] would increase because you have to write the description.

RACHEL ZHANG: Good, good. Yeah, OK. So now, we're going to run a CPR on U. You can have U of C because it's on the circuit-- this is U, instead of C. So issues. Maybe I should also write--

So issues. So one is that-- that Arnan pointed out-- which is that the input is now really large. So issue one-- input is large.

And if you remember from the GKR protocol, the verifier at the very end has to compute some points in the low-degree extension of the input. Before, we said, OK, the input is only size n. So they can compute it by themselves. It's only n times polylog time to compute it. But now the input also has C, which is size S. So now you have to actually spend time poly S to compute the final layer, which is bad. Like, we want the verifier to be efficient. Now we're back to square one. So we're going to deal with this. Any other issues?

AUDIENCE: Sorry. In the GKR protocol, the add times poly S's, do I only need to compute a couple of bits in the input layer, or do I need to compute the entire input layer?

RACHEL ZHANG: You need to read it all because what you're going to do is you're going to look at the input layer and then compute the low-degree extension. And the only way you can do something with the low-degree extension-- if you know everything in the coding. Otherwise-- yeah Yeah.

AUDIENCE: Sorry. I just have a question. Before you [? have ?] issues, is it obvious that the verifier can compute like add and mult tilde, given that you can add, but like--?

RACHEL ZHANG: Good, good.

AUDIENCE: Oh, is that the issue?

RACHEL ZHANG: Yeah. So--

AUDIENCE: [LAUGHS]

RACHEL ZHANG: Yeah. So Nikon's question is, how do you compute the low-degree extension of add and mult? Right now, I've only given you a Boolean formula to compute them. So you can compute the values on the hypercube, but you can't necessarily compute the logic extensions of them. So we're going to see how to fix both of these things. And hopefully, if I'm honest, these are the only two issues. So we'll see. Is this OK so far?

So issue one-- the input is now all of the circuit, plus the original input, so it's very large. So right now, this is what our circuit looks like-- is U. Big part is C And small part is x.

So here's when the logspace uniformity of C is going to come in. So if you ever read GKR statements, the theorem only holds for when C is logspace-uniform, which means that there's some small description of it. So some small log S size description of C.

And maybe let's call this description this. And this is Turing machine M-- that, basically, if it reads in this description, it can then output like the entire circuit C. And this Turing machine should run in logspace. But it can run for as much as S time.

So C is too big. Yeah.

AUDIENCE: So M is, like, fixed.

RACHEL ZHANG: Yeah, M is-- yeah, it's just a-- it's fixed Turing machine. Yeah. Yeah.

AUDIENCE: Just to make sure, intuitively, why this matters-- it's because C has to have, somehow, nice structure for it to be defined on inputs of different lengths and stuff-- being non-uniform?

RACHEL ZHANG: I think one main thing is that the verifier needs to be able to know what C is because like in order to get get on the circuit C for the verifier, they need to have some way of describing the circuit, in some sense. This is basically saying, the circuit has to be like log size so that the verifier can hold it and have some way of knowing what circuit C is.

AUDIENCE: I guess I'm trying to think about what kinds of circuits are excluded by this-- circuits that-- basically, you have to write out all-- the entire truth table to describe. Is that what we're trying to get rid of?

RACHEL ZHANG: So I think Ted is our local complexity theorist, so I'm going to--

AUDIENCE: This is the door I want. Basically, the circuit [INAUDIBLE].

[INTERPOSING VOICES]

AUDIENCE: OK.

AUDIENCE: You can't hardcode things, though. You can't hardcode a random string or something like that.

AUDIENCE: Right.

AUDIENCE: Yeah, [INAUDIBLE] circuit [INAUDIBLE].

RACHEL ZHANG: So C is large. And the problem was that reading this entire thing is too much. But we have a short description of C. So let's also, maybe, have a circuit that computes C. Maybe we'll call this M just because we have a Turing machine called M, but it's not a Turing machine. It's a circuit.

And this circuit will take, as input, the description C, with the things around it, and then it will output the entire circuit description. And that will feed into the universal gate universal K ?] that we just talked about. So this circuit here is our circuit C prime that we will actually be delegating on.

And just to be clear, which ones are the inputs? The inputs are this-- this description of z and also x. Like, this entire C here is now wrapped into the circuit. Yeah.

AUDIENCE: I put the notation-- like M, parentheses, bracket C equals C. So what do you mean by equal C? Like--

RACHEL ZHANG: Yeah.

AUDIENCE: --is there some other description of the circuit, or would say the Turing machine is able to run the circuit on any [INAUDIBLE]?

RACHEL ZHANG: So it will run on log S space and up to S time, and then it can output any gate of the circuit you want.

AUDIENCE: Any gate of the--

RACHEL ZHANG: Yeah, and maybe if you have an output here that's separate from its actual space that you can compute on compute on, ?] you can just output the gate by one.

AUDIENCE: Oh, OK. So you say equal C means find the gate.

RACHEL ZHANG: Sure. Yeah. I'll put in the description of C, or description of the gates of C So output gates. Yeah.

AUDIENCE: If you're just unrolling M, isn't that really deep?

RACHEL ZHANG: Good, good. OK, yeah. So what's the depth of this? So-- OK, M can run in time up to S. So this could, ostensibly, be depth S, which is bad because our verifier would have to depend on the depth also.

Well, it turns out you can do it in depth log S. And I'm not going to tell you how to do it. You're going to tell me by the end of class. So that's the goal, by 3:00 PM. And this will also be size S. So we'll get there. But it'll be your job to convince me. It's not my job to convince you. Cool. Anything else? Yeah.

AUDIENCE: How do you turn this Turing machine into a circuit?

RACHEL ZHANG: Good. OK. You'll tell me. It's the weekend, so the weekend don't worry. I think you'll be able to figure it out at the end of class. And then the other claim is that this is-- has easy add and mult gates. And also-- you'll also tell me that by the end of class.

So basically, the circuit is going to have all the nice properties you want. It's going to be depth, log S size S. The whole circuit-- now, C prime is going to be size poly S because it was also size poly S. This should probably be poly S, not just size S. And then total depth is going to be, like, poly log S. So-- yeah.

AUDIENCE: The statement is, like, truth table of M on C equals C-- on bracket C equals C. So do we need size of C different copies of M [INAUDIBLE] bits?

RACHEL ZHANG: Sure. Yeah, yeah. Sure.

AUDIENCE: [INAUDIBLE] all of them? OK.

RACHEL ZHANG: Sure. Yeah or you can think-- it's a thing-- you can have an output tape, where you can just output everything to, or another thing.

AUDIENCE: Yeah. Yeah. I just-- I want to make sure that it doesn't break.

RACHEL ZHANG: Yeah.

AUDIENCE: No, it doesn't break [INAUDIBLE] for me either. You can take it [INAUDIBLE].

RACHEL ZHANG: OK, yeah.

AUDIENCE: You might have just said this, actually, but [INAUDIBLE] on the line that says log S size description of C-- like, C in brackets. Like, C in brackets is a string. It's not actually, like, log S, right? It's, like-- S or something, isn't it? Or--

RACHEL ZHANG: No, this is going to be a log S size. So the circuit C is size S, but there's a short description that can tell you what circuit C is going to look like.

AUDIENCE: I guess we got a family of circuits-- like, C1 through CM. The description could be, like, the number n that you should print or something. Like a copy of that.

RACHEL ZHANG: Sure. Yeah.

AUDIENCE: Then why is that tree there? That log S-- wouldn't the number of leaves just be S? [INAUDIBLE] M tree.

RACHEL ZHANG: So this description is size log S, so this entire thing here is log-S size.

AUDIENCE: Isn't that a full [INAUDIBLE]?

RACHEL ZHANG: I guess it goes like this, more than-- yes, Alex.

[INTERPOSING VOICES]

AUDIENCE: Yes, because this is [INAUDIBLE] log S and [INAUDIBLE] size log S. It's roughly [? one ?] [? or ?] [? two ?] [? things. ?]

AUDIENCE: [INAUDIBLE] circuit that everyone could [INAUDIBLE].

AUDIENCE: There are conditions. [INAUDIBLE].

AUDIENCE: But we don't know how to unroll it yet. When we unroll it, we might [INAUDIBLE].

RACHEL ZHANG: Cool. OK, so the claim is given the circuit and given all these properties you can prove to me later about, then you can do GKR. So according to our-- I put it behind. I should push it forward. So the prover is going to run in time poly S, where S is the size of this new circuit, C prime.

This is size poly S. This is size poly. So everything is poly S, and poly of polys, polys of poly S for the prover. And the verifier is the depth of the circuit, plus the input length. Steps here-- this is going to be, like-- I think we said it was, basically, depth of the original circuit times log. And this is going to be really small. It's going to be, like, log S So it's going to be, overall, the same depth of the original circuit. And then the input size is now log S plus n. So it's not so much bigger either.

OK, input. Maybe-- yeah, OK. So the second issue that Nikon pointed out-- how do you compute the low-degree extensions of the add and mult gates, whereas we only currently have Boolean formulas that compute them on the-- I guess, in this case, it's even a Boolean hypercube. It's not even the h- hypercube. Yeah. Yeah.

AUDIENCE: [? Wait, I got another ?] [? question. ?] [INAUDIBLE] So the idea here was you wanted to ultimately get the bottom layer to be size small? But still, this middle layer that you get to that has C written on it when you run the algorithm. That's big. So at some point, you have to be able to get low-degree evaluations in this middle layer?

RACHEL ZHANG: The prover would do it. You would never have to do it.

AUDIENCE: I see. So the GKR-- the protocol we had last time was only for circuits with extensions that get smaller, right?

RACHEL ZHANG: [INAUDIBLE] Doesn't matter, because what-- the way I did is she extended every layer to size S anyway. So even if it wasn't S originally, she extended it to be big. So I think, sometimes, S can be viewed as an upper bound of each layer size, also. So second issue-- how do we compute add and mult? So we have-- let me just write down what I put up there. I can push this up.

So we have these two Boolean formulas. And it's going to compute add and mult. This doesn't have a go on top because it's not a low-degree ?] extension.

So, So this ?] is what phi of add looks like. It's a Boolean formula, so it's, like, a big tree. Not big, actually. It's very small trees. It's, like, polylog size.

I know output at the end. Yeah, what you're like-- yeah, out of b1 option something. So here's-- the inputs are all bits. So it's going to be 0, 1 to the-- I guess this should be taking as input. So the size of the circuit. And telling you whether or not that particular gate is add or mult gate, so it should be like size of like log n of the size of the C prime.

So this should be-- OK. Cool.

And our goal is to compute-- and so, here, we have-- H is going to be an H. And so here, M is log base H of S prime. So this is what we want, and this is what we have.

I want to repeat again. This is not the low-degree extension. This is going to be a low-degree extension. So what do we do for sharp set or sharpie Do you guys remember? Yeah.

AUDIENCE: Sorry. One more question. Is a low-degree extension-- but aren't low-degree extensions unique?

RACHEL ZHANG: So it's a extension that has low degree, but it's not that low. It's, like, somewhat low. It's low to [INAUDIBLE].

AUDIENCE: So we're talking about just extension [INAUDIBLE].

RACHEL ZHANG: Yeah. And it's not going to be the lowest degree, but it will have low enough degree We can call it low enough degree.

[LAUGHTER]

Yeah, low enough degree extension. So you can call this the LEID.

[LAUGHTER]

OK. So it's not going to be the low-degree extension, but it will be kind of. It'll be OK. So what did we do for sharpie the other-- like, the first class. This is, like, going three weeks ago. The idea there, we took a Boolean formula-- or a formula, right? OK, let me maybe jog your memory.

So the idea was you want to count the number of satisfied equations through some of CSP. You first turn this into some Boolean formula. And then you do something to it. And then you can apply the subject protocol. So what do you do in the middle?

AUDIENCE: Arithmetize.

RACHEL ZHANG: Arithmetize. Let's arithmetize is this thing. So maybe, first, before we arithmetize, we should change the input formats. So over here, we have h here. We have Boolean, like 0, 1. But what you can do is you can just take maybe a chunk of size-- sorry, this is a like really minor detail. I think-- it's fine. Let's do it anyway.

So this is going to be size-- log of size of h. And then-- so this is how many bits are going to be here. And you're going to take h. And you can convert it via some polynomial. It's going to be a degree h minus 1 poly. It's going to convert any element, h, of your bigger set, capital H, to 0 string. And you can do this a bunch of times.

So now we have this weird thing where you're feeding in h, and then you can use a degree like h polynomial to convert it to 0, 1s. And then now you can feed it into this Boolean formula. And for this part, we're going to just arithmetize it.

So with arithmetizing-- just to recall. So arithmetizing is you can take an and gate and replace it with the polynomial-- a plus b minus a times b. And then-- sorry, this is-- should be in, not add. And then OR of a, b-- is this right? No. It's the way. This is a plus b minus a, b, and this is a times b.

So every time there's a OR gate here, you can replace it by this polynomial. And if it's an AND gate, you can replace it with this polynomial.

So maybe-- there's another lemma that Yael proved during the first class that-- maybe it's been too long to remember, but maybe you can try to rethink about it. So the question is, what is the degree of the final polynomial you get here when you arithmetize it?

AUDIENCE: Is it 2 to the depth? [INAUDIBLE]

RACHEL ZHANG: Yeah, yeah, exactly. So it's a number of input gates here. I think that's the way she said it in the first class. It was just a number of input gates, times-- I guess, here, you had to times by the degree of each thing you're feeding in, which is to be h minus 1. So it's going to be-- log S prime, which is the number of gates in the bottom, times-- let's just say h instead of h minus 1 because it's a little shorter.

So this is going to be the polynomial that we're going to use for add tilde and mult tilde. So it's not the low-degree extension but it's a good thing. Yeah.

AUDIENCE: Is this is over-- just to be clear. The extension is over-- is that right, or is this over [INAUDIBLE]?

RACHEL ZHANG: Yeah, so I guess what's happening is you're first converting your h over-- yeah. Sorry. Yeah.

AUDIENCE: The way you've drawn it, it makes it seem like you're converting h into log h bits. What if I just want to convert it to a single?

RACHEL ZHANG: So h is big. h is, like, 2 to the log h sort of information. Or-- maybe it's the wrong information. So h comes from this bigger set H. So you want to convert it to 0, 1 so you can feed it into the Boolean formula. Otherwise, it's not type-compatible.

AUDIENCE: But the Boolean formula only takes in-- like, when you fix a Boolean formula, it has a bunch of inputs that are bits. But each bit-- you called them [INAUDIBLE]. So how do you plug in log h bits into an input that only have one bit?

RACHEL ZHANG: OK, Yeah. So you can convert this into a number of bits. It's going to be log h many bits.

AUDIENCE: The first log n [INAUDIBLE].

AUDIENCE: [INAUDIBLE]

RACHEL ZHANG: Yeah, so the conversion isn't from one h symbol to one bit. It's going to be one h symbol to many bits.

AUDIENCE: To many bits.

RACHEL ZHANG: Right. Many is going to 1 h bits.

AUDIENCE: I thought we just wanted to find a big argument in circuits is that when you evaluate it, when the h and r is 0 and 1, it's equivalent to the original.

RACHEL ZHANG: OK. Yeah, sure. So if you want to keep this-- this in 0, 1 so you can have big set here H just be 0, 1, then this is just identity. You can just plug it straight in. I guess this conversion is-- I don't know. Yael called the thing based on whether H is like a bigger set. So I've just been converting it to H is pretty much that, basically.

AUDIENCE: Oh, OK h is different than f.

RACHEL ZHANG: Yeah.

AUDIENCE: OK.

AUDIENCE: So you're literally just saying arithmetize, right?

RACHEL ZHANG: I'm literally just saying arithmetize, yeah.

AUDIENCE: And it's still a deterministic procedure. So you'll get to some unique point, given the same [INAUDIBLE] point, but it's just not the low-degree extension.

RACHEL ZHANG: Yeah. But it's low enough degree, which is going to be the number here, because if it was a really low degree, it would just be h degree. But this is a log factor bigger. But that's going to be OK for us because, as a verifier, that's fine, I guess. Yeah.

AUDIENCE: I'm still a little bit confused about-- so b add basically computes for one gate, right? That essentially like computes-- given one gate it simulates the circuit for one particular gate, right?

RACHEL ZHANG: No. So this formula-- so the input is going to be any particular location in your entire circuit, and it's going to compute this log depth-- this polylog-size formula to tell you what that gate should be-- whether that was the add gate or not. So the input is not the specific gate. It's going to be any of the gates in the entire circuit.

AUDIENCE: Yeah, so it's one formula for the entire circuit.

RACHEL ZHANG: Yeah, one formula for the entire circuit.

AUDIENCE: But then why are you-- and so where exactly-- what-- we have these h's. And that means that we have, technically, I guess the same number of inputs. But we have blocks of inputs now. And how exactly does that change? Because that means that we have-- technically, we're combining-- it just feels weird because, in some sense, it feels like we're mixing up information because we're saying-- we're assigning one thing-- one thing in a large space to a bunch of things in the Boolean space, of the [INAUDIBLE].

In the large field-- h is in the larger field, right? And we're saying that we're just assigning one h to this spread of things in the [INAUDIBLE] field. And it just feels-- I'm not [INAUDIBLE] a lot [? of things. ?]

RACHEL ZHANG: OK, yeah. So maybe the confusion is, like, this part is 0, 1, so it's really just Boolean. And then over here, we have this weird h going into it, which is not Boolean. So, what is this polynomial doing with all this Boolean stuff? So that's part of the question? Yeah, so maybe we should think about it as in-- Let's first arithmetize this circuit here. So we're going to get something over a large field already. And now we're going to plug in these polynomials into the arithmetized thing, which is already going to be polynomial. So we're-- it's like polynomial composition.

AUDIENCE: But the thing is you're not doing one to one. You're doing one to many.

RACHEL ZHANG: If you think about it, the amount of information is conserved because here, there's H options-- capital H options-- for this h1. And then when you plug it up here, you now still have only h options for the resulting 0, 1 string.

AUDIENCE: Well-- I mean, yes, but it also-- it felt that-- yes, but in the higher field, we only have-- now we only have the log S prime different inputs from the larger field. Now we're jamming them into this-- or rather, we only have a few-- we have much less things-- the input from the larger field, and we're jamming that in.

And it just-- each element is getting is covering a bunch of bits, and those might [INAUDIBLE].

RACHEL ZHANG: So it turns out that there is a unique mapping from the string of h1 up to hm. And then the 0, 1 to the log S prime string. And so it's a bijection. You can go back and forth between them. So it's actually-- there's no clashing. It's going to be just one to one.

AUDIENCE: OK. I'm sorry. I'm just not following the bijection. [INAUDIBLE]

RACHEL ZHANG: Sure, OK. Yeah.

AUDIENCE: I guess that's where my [INAUDIBLE].

RACHEL ZHANG: Sure. OK, so what's going to happen is-- maybe let me zoom out of that. It's a little messy. So let's say you have-- this is going to be just-- because this is going to be our 0, 1 layer over here.

AUDIENCE: I'm sorry again for--

RACHEL ZHANG: Yeah, no, you're fine. It's good to ask questions. So this is going to be log h number of bits. And the idea is we're going to take some h, and we're going to map it here to log h many different bits. So here-- one way you can do this-- and you can write this in binary. So maybe this-- maybe h here is, like, size 8. So this would correspond to a binary string.

AUDIENCE: I get you can do that. But I'm like, why are we allowed to do that? And how exactly-- why exactly is this helping us? I understand that you can write it in binary on only that. How exactly is this helping us again? Because it feels like now each element-- each big element in the extension-- like, before, we were just saying that we're going to extend each individual-- we're going to assume that 0, 1 values are actually now extended values.

But then, now, we're doing it in such a way that-- because before, we were assuming these values from the large circuit are-- we're going to say-- if happen to have, obviously, an AND gate, then we're going to have one big value and another big value, and then multiply them together [INAUDIBLE].

RACHEL ZHANG: OK. Can you hold that? How about if we take a break now, and then we'll come back in five minutes, and then we'll do the second half of the class? I think that's all for this part. And maybe we can talk offline.

AUDIENCE: Sorry.

RACHEL ZHANG: No, you're totally fine. Let's talk about it then.

[INTERPOSING VOICES]

So how many of you guys know the theorem exists or what it says or anything?

AUDIENCE: Somehow, I already forgot.

RACHEL ZHANG: OK, it's fine. We'll talk about it again. So this is a really famous theorem. It's a great theorem. It's very good. So we should talk about what these things are. So can one of the local complexity theorists people tell you what IP is? It doesn't have to be the local complexity theorist. It could be any of you. [LAUGHS]

AUDIENCE: Positive statements about [INAUDIBLE] the interactive protocol between polytime [INAUDIBLE] verifier and the computationally-bounded prover.

RACHEL ZHANG: Unbounded prover.

[INTERPOSING VOICES]

Let's see. OK. Yeah. So-- OK, so IP is a class of all statements that have interactive proofs. Interactive. That's right. OK, yeah. And the prover can be unbounded time. And then the verifier's running on poly time. So the interactive proof should also be poly size. Otherwise, the verifier can't read it.

And then what is PSPACE?

AUDIENCE: Polynomial space-bounded TM languages?

RACHEL ZHANG: Great. OK. Polyspace computations. Yay. And so the easier direction-- so it was shown in 1992. These two things are actually equal. So the easier direction to show is that IP is in PSPACE. The way you do that is by simulating all possible transcripts, and then you can do some counting stuff. We won't go into the details.

But in 1992, it was shown the other direction, also held. So this is by Shamir in 1992. So in fact, PSPACE is also in IP, which means that these two classes are actually equal.

And to do this, he constructed an interactive proof for any PSPACE computation, and that would give you the inclusion. So in particular, this means that any PSPACE computation has interactive proof.

So going back to the concept of delegation, which is what this class is entirely about-- how do you delegate a computation to a bigger prover, one you can run very quickly? Well, this is delegation, right? OK. It's basically like, I want you to compute some small space computation, like polyspace. And I can ask the cloud, hey, can you run this awesome Shamir interactive proof for me and just tell me-- prove to me that you did it correctly? And we're good, right?

So what's the issue? Why aren't we done? Why did we do GKR at all? The reason is because, here, the prover runtime is 2 to the S squared. So in PSPACE-- any computation is doable in time 2 to the S. But here's this additional-- so this is S2 that's squared. So basically, the time to prove is much more than the time necessary to simply compute, which is why there's all these follow-up works and why this is actually so-- an interesting question today.

And there are some follow-up work. I don't know the reference, and Yael didn't either. And I don't know if memory is wrong either. So it could be wrong. It could be true. So it's a possible statement.

[LAUGHTER]

So it's possible that there's a follow-up work that showed how to have a proved runtime of 2 to the S log S, which is significantly better than 2 to the S squared. But it's still not 2 to the S, which would be the best you could hope for any arbitrary PSPACE computation. So today, what we're going to show is-- so today, the theorem is going to be as follows. So assuming GKR, which is an unconditional statement. It's not an assumption. It's just-- this is what we're going to use in the proof.

So since we have GKR, we can get IP equals PSPACE with-- time 2 to the-- I should probably right poly 2 to the S prover.

So what this means is-- so for, maybe, the hardest PSPACE computations, assuming that, I guess, the hardest thing you can do is time 2 to the S, then we do have actually efficient delegation protocols for these things. And they're, in fact, also proof systems. So they are, like, information theoretically secure.

And maybe I'll just mention one open question is, what about general time TSPACE-- space computation? So-- where T's not necessarily S, can you do an interactive proof as efficient?

I think this is not known. So if you want to think about some proof systems, this is a question you can think about. Are you guys OK if I raise this? This is probably fine, right? OK.

AUDIENCE: Here, efficient is, like, poly T?

RACHEL ZHANG: Yeah, poly T. And you can definitely ask more-- even stronger questions. Can you do it with linear time overhead, additive, whatever, overhead? Yeah, there's many questions.

OK, so how do we describe a PSPACE computation? Someone said that it's a space of all computations that are doable by a Turing machine with bounded space. So let's say that we have our Turing machine, M. And let's say it runs in space, S. And maybe I could say, on input 0, 1 to the n.

And maybe the thing we want is-- and maybe we can just say time-- and also time, T. And I guess I could set it to 2 to the S, just for now. Is that OK? So let's say that maybe we have a Turing machine and we want to see that within time T, which is 2 to the S, that eventually, it will get to the output, like the accepting configuration, or not.

So by this point, it should get to accepting configuration.

So here's a trick. The idea here, we're going to use GKR to prove the theorem. So we're going to have to construct some circuit to delegate some computation with GKR with. So this Turing machine here-- it runs in space S-- can run time-- up to 2 to the S. So if you naively do this, you would be [INAUDIBLE] depth 2 to the S circuit. That's not so good. We want it, really, to be depth S.

So the thing about let me write that down. So the goal is to convert to depth S. That's the goal.

So here's the trick. The trick is as follows. We're going to look at the transition matrix of all the possible states of the Turing tape, basically. So we're to have a bunch of different possible states here and a bunch of different possible states here. So how many possible states are there? Well, there is 0, 1 to the S. That's what the state could actually say on the tape. But that's also where the Turing machine is right now.

So maybe times the index and S should say where the Turing machine is right now. And then it would do one computation, move somewhere, and then it'll be an adjacent place at a different state. And then, now, the new state will be like a slightly different string, and it'll be-- the Turing pointer will be at a different location. So here is also going to be the same states.

And the point is we're going to put a 1 at a location if there's a way to get from this state here to this state here in one time step. And 0s everywhere else. Yeah.

AUDIENCE: Does this state include the internal automaton?

RACHEL ZHANG: Yes, that's a good point. It's actually a very good-- yeah.

AUDIENCE: So there should only be, like, one 1 in each row.

RACHEL ZHANG: Yeah, there should be one 1 in every row because the thing you were going to do after one time step should be deterministic based on your current location-- your state based on your-- your internal state of your Turing machine, as well as what you see around you. I'm using this as just the number of states in the Turing machine. Anything else I'm forgetting within these states? There's just a bunch of information that I don't remember.

So anyway-- so I guess-- the key point here is that this matrix is actually very easy to compute, because given maybe a state here and a state here, you can compare very-- in very low depth, very efficiently. So it's a very uniform description of this check, or of the circuit that will generate, whether or not the particular corresponding matrix value is a 0 or 1.

And that'll be important for us because we want-- we want a uniform circuit, I guess, at this point. And this will make it uniform.

AUDIENCE: So [INAUDIBLE] previously.

RACHEL ZHANG: Yeah, OK. So the matrix here is easy to compute, is my claim. The reason is you can take any state here, any state here. You can do some checks. It's a very-- it's a small formula. You can just check it very easily by knowing what M is, by knowing what you see. And you can tell if there's a transition from that to the next state. So the matrix here is easy to compute, is my claim.

So maybe let's call this matrix A. So what I just said is that Aij-- this is going to be the ith row in the jth column-- entry of A. This is going to be a 1 if-- so state i goes to state j in one time step.

And so that's good. What we really want to know is whether i will go to j or the starting configuration will go to the accepting configuration in T time steps. That's what we want.

So here, we have a copy of A. Maybe let's just do some computations just for fun and see what happens. So you have a copy of matrix A here, a copy of matrix A here. Just multiply them via matrix multiplication. So we have A squared. What is the value of the entry in the ith row and the jth column of A squared?

AUDIENCE: [INAUDIBLE]

RACHEL ZHANG: Yeah, so-- I guess I'll write here, actually. So A squared-- so the ijth entry of A squared is whether i goes to j in two steps. And maybe for those of you who haven't seen this before, let's just do it so you see why.

So if you want to look at the ijth entry of this product of these two matrices, what you want to do is you want to take the ith row, look at all the entries across it, take the jth column, look at all the entries down it, and you want to take the pairwise-- you want to take the product and the sum. So you want to take this entry times this entry, this entry times this entry, and so on.

So what's essentially happening is, here, you want to take-- you want to look at the entry corresponding to going from i to k. And then you go from k to l and k to j. And then you do this for k equals 2, also. We do it for-- llike you go from i to k. And then you'll do it from k to j.

And so the final entry will-- the sum of all of this-- these products-- will be 1 if and only if there is a way to go from i to some k to j in two time steps. That OK? Or-- I don't know. OK.

So in general then, if we want to look at the transition matrix of having-- after going for T time steps-- you keep doing this. And you get that the ijth entry of A to the power is whether i goes to j.

AUDIENCE: So here, [INAUDIBLE] whether [INAUDIBLE] exactly T steps? Like, no more, no less?

RACHEL ZHANG: Yeah, OK. That's a good point. So maybe we can say this should be in exactly T time steps. And maybe you can have your Turing machine stay as accepting configuration if it's already reached it, maybe, for instance. That's one thing you could do. Another way you could do it is you-- you could change this matrix because it's a one-to-one diagonal so that you can always waste time if you want. But whatever version you want is fine. Just-- maybe, let's say for now, after exactly T time steps, it should reach the existing configuration. And just have it start accepting place.

So the point is if we can compute what AT of the input configuration, comma, the accepting configuration is, we know if this PSPACE computation is accepting or not. That good?

So what we want is-- OK, so we're going to write a circuit to compute this. Is it OK if I'm very loose about this? I just feel like all these things about exactly how everything ends up is just terrible. I don't know.

OK, so I guess one thing you could do is you could just compute and then A squared, then A cubed and A to the fourth. But it's not so good because you can do it for T times your circuit C, depth T. So what should you do instead?

AUDIENCE: Repeated squaring.

RACHEL ZHANG: Repeated squaring. Great. OK. Yay. OK. So the idea is we can compute this matrix A So this is going to be very easy to compute because we talked a little bit about how every entry of this matrix is-- you can compute it very easily just by looking at the two configurations are that you're interested in.

And then you can compute A squared from this by just-- I guess you do one division, one multiplication. And then we can do A to the fourth, A to the eighth and so on. And eventually we'll get to A the T, which I don't have room for.

This is going to be our matrix-- or our circuit. Let's look at the depth-- what the depth of the circuit.

[INTERPOSING VOICES]

OK, so it's, like, log T-ish.

AUDIENCE: But each, like, thing--

RACHEL ZHANG: Yeah. OK. Times something.

[LAUGHTER]

AUDIENCE: But don't you have to do-- you have to add up T thetas to that thing. But you can make it a binary tree, so that will have [INAUDIBLE]. Yeah. OK.

[INTERPOSING VOICES]

AUDIENCE: Poly S.

RACHEL ZHANG: We're going to say poly S. It's probably, like, log t times poly S anyway. So that's pretty good. Means that it's a verifier when running this computation with the prover in GKR. It's a scale to the depths. The depth is now in poly S. What's the size of the circuit?

AUDIENCE: Poly S-ish, [INAUDIBLE] 2 to the S.

AUDIENCE: [? Just ?] [? write ?] [? it out. ?]

[INTERPOSING VOICES]

RACHEL ZHANG: Yeah, so I guess the matrix stuff is going to be size 2 the 2, or something times 2 to the S.

AUDIENCE: 2 to the S squared.

RACHEL ZHANG: Yeah. 2 to the S. So poly 2 to the S. 2 to the S times 2 to the S is 2 to the 2S. So yeah. Yeah. OK. [LAUGHS] OK, so if we apply GKR to this, what do we get? So let's run GKR.

Oh, and at the very top, we need to, then, look at the input state and then check the corresponding entry. So you want to read this particular entry, maybe. But you can read it by doing some of pretty small computation on what the input and accepting computations are. So let's run GKR. So prover runtime?

AUDIENCE: Poly 2 to the S.

RACHEL ZHANG: Yay. OK I'll let you do this. Verifier runtime?

AUDIENCE: Poly S.

RACHEL ZHANG: Yay. OK I guess that's what we wanted to prove, huh? OK. Nice.

[LAUGHTER]

Good work. Are there any questions about this? Or is this-- does it make sense? Yeah.

AUDIENCE: Sorry, I might have missed an important point, but where is the cost of actually doing the squaring?

RACHEL ZHANG: I guess a really good depth in the circuit is each square you do is going to take some more depth of your circuit.

AUDIENCE: Yeah.

RACHEL ZHANG: Is that what you're saying?

AUDIENCE: Yeah, that's the question.

RACHEL ZHANG: Maybe I didn't understand the question.

AUDIENCE: Oh, is it-- what is the cost of squaring A? Just going from A to A squared, what's the cost?

RACHEL ZHANG: Sure. Yeah. So how do you compute a squared. You can do every row and every column. And then what if I add them?

AUDIENCE: Oh. So is it-- you're just doing--

RACHEL ZHANG: You're just doing matrix application.

AUDIENCE: I see.

RACHEL ZHANG: So maybe you can tell me how [INAUDIBLE].

[LAUGHTER]

[INTERPOSING VOICES]

AUDIENCE: But are we supposed to know how that thing that you said we need to prove to you-- are we supposed to know--

RACHEL ZHANG: Oh, yeah, that's a great question. So earlier-- I already erased it, I think. Wait. Are there any more questions about this before we go to that?

AUDIENCE: So we're asking you this to prove that because [INAUDIBLE].

RACHEL ZHANG: You can't use IP. Just use the logic you just had.

AUDIENCE: OK, OK.

RACHEL ZHANG: Yeah. So how do we construct the circuit now?

[INTERPOSING VOICES]

AUDIENCE: So that's another reason we need logspace uniform [INAUDIBLE].

AUDIENCE: It runs in logspace. That's a Turing machine. So [INAUDIBLE]

AUDIENCE: Yeah.

RACHEL ZHANG: Yeah, OK. So this is basically a scaled-down version of that. This is logspace uniform. So you're running in logspace. Then you're running in time space S. So this is a exponentially smaller version of that. So all the parameters over there-- you scale them down. So we have the depth over here. Was poly S. The log of that is log S. The log of the size here is S So we have all those things we wanted.

AUDIENCE: How do you incorporate that input into this-- you need to know what the starting state is.

RACHEL ZHANG: With this one or with [INAUDIBLE]?

AUDIENCE: With that one.

RACHEL ZHANG: The input state. Yeah. So I think the way I normally think-- or I like to think about it is you only check the input configuration at the very end, once you complete the entire thing. So then you only have to read the input, comma, accepting configurations entry. And there you can just do a formula to read off what it says and then look at the corresponding value.

AUDIENCE: So almost the entire circuit is just-- doesn't even read the input.

RACHEL ZHANG: Yeah, it's only the very end.

AUDIENCE: But, I mean, this is very accepting, and that one doesn't really accept it in print.

RACHEL ZHANG: Yeah.

AUDIENCE: [INAUDIBLE]

RACHEL ZHANG: Yeah, yeah. OK. So maybe-- yeah, so I guess each configuration corresponds to-- it's a configuration. So there's some data about what the gate's output-- gate value is or something. Maybe you can read off the value there.

AUDIENCE: Each entry, [INAUDIBLE] [? comma-- ?] is the gate here [? or not, ?] and then each entry [INAUDIBLE]?

AUDIENCE: And this sounds like an easy add, mult situation, right? Because [? it's just ?] [? computation? ?]

RACHEL ZHANG: Yeah. The important point here is that computing the matrix A is very easy. That's, really, the key thing here. And that's also why, when we said what-- we were behind. But when we said what A looks like, it really had to be also where the Turing pointer was, because if you didn't know where the pointer was, it could be very large. Yeah.

AUDIENCE: [INAUDIBLE]

AUDIENCE: So you can factor out what you're saying into the form of-- if you have a low space computation, you can turn to a low-depth circuit. Is that a is that a fair [INAUDIBLE]? that you've used?

RACHEL ZHANG: Yeah, it's actually a kind of interesting statement, huh? I don't know. I didn't know that you could do this until recently either. So I guess-- the interesting thing here is instead of being a really tall and skinny computation, you can turn it into a circuit that's really fat and short.

AUDIENCE: That's why we're getting [INAUDIBLE] depth, not [INAUDIBLE].

AUDIENCE: Wait. Sorry. OK. I was just-- in terms of where we're at in class. So we just proved this theorem. And is this the same reasoning for why the logic conformity thing works out, too-- by the same [INAUDIBLE]?

RACHEL ZHANG: You tell me.

[LAUGHTER]

AUDIENCE: Yeah. You tell me. I'm delegating.

[LAUGHTER]

AUDIENCE: It's kind of weird because you don't actually take A as input. A is [? hardcoded ?] into the circuit, right? That's why the input can still be the small thing. That's just describing the starting state.

RACHEL ZHANG: Yeah, The Turing machine, M, is what determines what the circuit action looks like. So maybe as part of the input of this, you're also going to have a description-- like, the Turing machine in its entirety. The Turing machines are finite size, so it doesn't matter.

AUDIENCE: Yeah.

AUDIENCE: And that's why we don't need to worry about complicated add, mult predicates inside, because it's actually just looks like this.

AUDIENCE: I'm still kind of weirded out by the fact that all of that-- until the star at the top, all of this is not even dependent on the input. So it's like-- in this case, it's just going to be constant. Like, [INAUDIBLE]. So then why are we going through the trouble of computing all these layers, instead of just using the top layer, where we--

RACHEL ZHANG: I mean, how do you hold the top layer?

AUDIENCE: So you're saying that it's easier to encode the entire computation thing than just the result.

RACHEL ZHANG: It's easy to say what you have to do to compute it than to actually compute it.

AUDIENCE: [INAUDIBLE]

AUDIENCE: [INAUDIBLE] M, not C. M [INAUDIBLE] C.

AUDIENCE: Or brackets used a compressed representation of C. [INAUDIBLE]

AUDIENCE: But I'm saying, why don't you just hardcode it in the top layer of M?

AUDIENCE: That would [INAUDIBLE] C.

RACHEL ZHANG: They didn't even [INAUDIBLE]

AUDIENCE: Well, it's just-- I mean, it's just a string of size of C. So you can't hardcode [INAUDIBLE] solution.

AUDIENCE: OK. It's weird, but I can kind of see.

AUDIENCE: So basically, we went through this trouble and we were able to go through this trouble because we assume that C can be described succinctly, like logspace succinctly.

RACHEL ZHANG: Yeah.

AUDIENCE: Is poly log S uniform [INAUDIBLE]?

RACHEL ZHANG: I'm going to delegate-- push it to somebody else. I don't know how log blows up. Probably not, because if you take-- the transition matrix A you have is going to be 2 to the description size or whatever. Like, the space. So if you have larger space, it's really bad.

AUDIENCE: But space S gets turned into depth like S squared, roughly, in this transformation.

RACHEL ZHANG: Yeah.

AUDIENCE: So if we have space S squared to start, we can definitely [INAUDIBLE].

RACHEL ZHANG: Yeah, but the question is the prover runtime, right? But the reason why we're doing any of this is because the prover runtime. You actually get an advantage in the prover runtime. So we're done early. So maybe we can all go get some corn. And-- yay.

[APPLAUSE]