[SQUEAKING]

[RUSTLING]

[CLICKING]

PROFESSOR: Let's go back. So I want to say-- before, actually, I tell you that-- just a high level idea of the other proof, I just want to-- I got a very good question in the break, so I want to emphasize-- I want to answer it for the class.

So the question was the following. When I talked about interactive proof I said sound is half. And if you want more than half, do repetition. Do parallel repetition. And now when I talked about argument I went to negligible. So the question is, why don't I do-- why is the change of language, of terminology, why for proofs I said half and you can do repetition? And for argument, I immediately said, no, I want it to be negligible.

And the answer is actually pretty weird. The answer is that, for proofs, we can do-- we can repeat-- if you have an interactive proof and you repeat it in parallel, let's say, lambda times, then the soundness goes down exponentially with lambda. If it was half now, it's 1 over 2 to the lambda, like you would expect. With arguments, that's not necessarily the case.

Once we go to the land of crypto, things become a little weird, and it's not the case that any argument, if you repeat it in parallel lambda times, if you had complete-- if you had the soundness half, now it's soundness 1 over 2 to the-- 1 over 2 to the lambda. That's not the case anymore.

Now, we have examples that show this is not the case, that actually soundness doesn't go down. But these examples are very contrived, very, very contrived. It's not a natural argument. My guess is for any natural argument, however you define natural, soundness probably does go down exponentially when you repeat in parallel.

Now-- but it's not true all the time. So actually, this parallel repetition, you can rely on it. Now, we can do sequential repetition. And then the soundness does go down. If you do once and then you do the-- so you do the interactive argument once. You are done. Do it again, done, do it again, done. Do it lambda times, and accept if and only if all of them accept, then soundness does go down exponentially, like you would expect.

However, then you increase the number of rounds by lambda. So you went from constant maybe to nonconstant. So there's a price. So it's not like for free you-- in the interactive proof setting, it's like for free. Just repeat in parallel. It's like for free. Here, now the round complexity changes. It's not very free anymore. So that's why we put here negligible. That just explained the change of language, why here we chose negligible before we chose 1/2, or 1/3, 2/3, whatever. The constant didn't matter because you can amplify.

OK, questions before I go back to say a few words about-- so the plan for our the hour, less than an hour we have left, I want say a few words about the Barak-Goldreich proof called-- they call it "Universal Arguments," that paper. And then we'll move on to construct hash functions. Yeah?

OK, so let me just say a few words. I don't want to go into details about the Barak-Goldreich, but just to tell you the high-level idea. This proof, what it did, what it tried to do is to say, look, if there are no collisions, I'm going to construct an entire PCP. I'm going to try as much as I can. And now I get a contradiction because there is no PCP, and hence, I conclude that there was a hash function. But for this, I needed to construct an entire PCP and, therefore, run in time that's proportional to the PCP size.

OK, instead, the Barak-Goldreich proof, what they say, they say, let's look at each kind of query separately. So in other words, the PCP, let's say, has length m. Let's look at the PCP at location i. And I asked the following query, the following question. Here's the question they ask.

In location i, if I run the PCP, I have a cheating prover here, yeah? If I run him twice or many times, where all the times have location i in their, one i, not all the PCP, for a certain index i, let me run the PCP verifier-- let me give the cheating prover like a randomness from the verifier so that the queries include index i. And let me take many of them. Is there a collision in i, or is all the time he answers the same way. That's the question.

So for an index i, does he always answer with the same bit, assuming he answers correctly? If he doesn't answer correctly, if he's rejected, then we throw it out. But all the time that he answers correctly when asked location i, does he answer consistently with the same bit or not?

Now, if he's answered, if not, if some probability, I don't know, epsilon he answers with 0, and probability epsilon he answers with 1, we found a collision. And then we're happy. If not, we didn't find the collision. Now they ask, so let's call an index like good if there's no collisions on that index. Then he behaved like-- the prover behaved well. He behaved good.

So for every index I ask, is it good? In the analysis, I don't yet-- I don't have an adversary yet. I'm just saying, look, there's a cheating prover. For every index, let's look at the probability that when he answers correctly, he answers always with the same bit. Let's say we answer the same bit with probability-- he answers correctly, but with two different answers with probability greater than, I don't know, epsilon to the third, whatever, some function of epsilon that's non-negligible.

So it's good if he doesn't do that. He answers consistently almost always. And it's bad if he answers inconsistent with probability at least, I don't know, epsilon to the third. And now what they say is, it must be that many i's are bad. If almost all the i's are good, a significant fraction of the i's must be bad. Namely, there should be collisions on them. Why? If almost all of them were good, then and the good ones I will-- in the--

OK, so first they say many of the indexes i must be bad. Once you establish that, then you just break collisions by choosing a random i, hope it's bad, and you're done. You're not doing many i's like we did. Just one random i with some good probability or bad. If it's bad, you found a collision, done. OK, but one can say maybe there's-- maybe the number of bad is like 1 over n. Maybe you need to repeat it n times until you find a bad one.

So they say, no, the number of bad ones need to be like polynomial in at least non-negligible in lambda. Why? If almost all of them were good, where there's no collision, then you can-- OK, and these reconstruct the PCP. And the good ones do the same kind of idea of reconstructing the PCP, but only on the good indices. Then they show that if you have enough good indices, then you'll be accepted with non-negligible probability.

So there must be enough bad indices, indices that the cheating prover enters inconsistently. Once you have that, then now, once you establish that in your head, that we understand that, now here, the adversary just chooses one random i. He runs the PCP verifier twice. He generates two randomness that generates two fresh index set that include i.

So they had one assumption that they needed about the PCP, that given i, it is easy to find r such that the PCP verifier and input r outputs a set that includes i. And turns out all PCPs we know, or all PCPs I know, satisfied this property. So that's-- it turns out it's a natural property. It's easy given i to find randomness, true randomness that generates a set of queries conditioned on i being in that set.

Once you can do that, then he chooses two R's, randomness that have i in the set, and hope for collision. And then you can argue that you get collision with non-negligible probability because non-negligible fraction of them are bad. And the bad ones are defined so that when you choose two random invocation that include i you find a collision. So they found a collision.

So they just did it. The difference is they did it index by index. They said, does he find collision? I have a cheating prover, p star. Does he find the collision on index 1? Does he find the collision on these 2? This isn't an analysis. Once, and then they establish, he must find collisions on many indices. And now, they're ready to break, the collision, the hash function. They just choose a random i and break by running the cheating prover twice.

So it's just a more efficient way of breaking it. Instead of the prover running in time n, their prover, instead of the adversary running in time n because they-- in this proof, we ran the cheating prover n times or size of the PCP times, at least times 1 over epsilon, they run it only security parameter times divided by epsilon with polynomial overhead. But independent of n is the point.

AUDIENCE: And that's what makes the argument universal?

PROFESSOR: OK, good. You're asking what's that related to universal? Yeah, because the reason they call it universal is because-- yeah, I guess the fact that it doesn't depend on n is-- because they talked about the universal language that has a Turing machine x and t, and it's in the language of those because it's a witness for-- this Turing machine runs in time t, and they didn't want things to blow up with t like they want. So this universal language what they did-- so they want one hash function that works for this language. And t now is given in binary in this language. So how can you have one hash function, one universal hash family? And that's why the name universal.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah.

AUDIENCE: What is the name also of the paper--

PROFESSOR: The name of the paper is Universal Argument. I'll link to it in the website. Yeah, it's not there yet. I only linked to Micali but I'll link to it. It's a really nice-- it's a really, really nice-- especially the overview explains it very nicely. It explains the proof I gave here, and it explains their proof in a very nice way. So I'll add a link.

OK, any other questions before we go and construct the hash function? So let me-- oof. This is what I want to construct. But any questions before we go ahead and construct it? OK, let's do it.

So what I'm going to do is I'm going to first construct a hash function that's not quite what I want, but it's going to be a building block to get what I want. And this hash function-- first, it doesn't have a local opening. So it doesn't satisfy that. And second, it's not even goes from 0, 1 star to 0, 1 to the lambda, which is what I want. It actually goes from 0, 1 to lambda to 0, 1 to the lambda.

So just shrinks a little bit. It doesn't shrink by an arbitrary amount. It just shrinks by a factor of 2. That's it. And it doesn't have local opening. So it's a bit of pathetic, but we'll do that first. And then we'll show how to use that to get what we want. So the first-- so now we're going to talk about constructions.

So let me first show you a construction based-- so all of cryptography, in particular, the hash functions, are based on assumptions, an assumption that some math problem is hard. And we know how to construct collision-resistant hash functions from a bunch of assumptions, many, many assumptions. I'm going to show you one that's just very simple and based on the discrete log problem.

So the discrete log problem just says that there is some group. You can think of zp star, so everything multiplication mod p. But in general, there is a group such that-- so this is the discrete log assumption. This is what we're relying on the hardness of this problem that says that given a group G of order, order meaning number of elements, let's say, 2 to the lambda. So it has 2 to the lambda elements. So I have a family of groups. And given a generator, G, so every element-- think of multiplicative groups, OK? So this group is like G to the x.

So for all the group is of form G generates the entire group. The assumption is-- so given a group of order-- so it's a finite group with 2 to the lambda elements, so more or less to the lambdas. And the assumption is that given a random, given G to the x for a random x, take a random x in the group, like in the size of the group, given G to the x, it's hard to find x. That's the assumption.

So in other words, for any poly lambda adversary A, the probability-- there exists a negligible function, the probability that A-- A knows G and P and the group. These are kind of fixed. The generator of the group is fixed. He gets G to the x for a random x. And the probability that he can find x is negligible.

And usually, we think of these groups as-- an element of the group is described by-- so any group-- any element of the group you can describe as an element for every H and G, you can think of H as being in 01 to the lambda. You can describe it as efficiently in lambda bits. And just think of zp star if you want as-- all the elements mod P with multiplication mod P. That's an example. So this is the assumption.

Now, some groups this holds. Some groups it doesn't hold, OK? For example, you can think of it as in a group, in the additive group, mod P, this is false. But in the multiplicative group, we believe that it's true. Is it really true? We don't know. We don't actually know. We believe it's true, and we base cryptography on this assumption. So we base cryptography on the fact that in zp star, if you take elements in the multiplicative group zp star mod p, then if I give you a random-- if you take a generator of this group and you raise it to the power of a random element, if I give you a random element in the group, essentially, then finding the discrete log we believe is hard. Yeah.

AUDIENCE: Do we know if quantum-- there's a easy way for quantum to break--

PROFESSOR: Good, yeah, so OK, I say we believe it's hard. Why we believe it's hard, I have no idea. I think the only reason we believe it's hard is because we don't know how to break it. That's really it because quantum computers can break it. They're subexponential algorithms. And if we didn't think we were that smart, we would have think it's easy because every evidence we have is that it's not that hard. But because we don't know how to break it and we think we're pretty smart, then we--

But no, the truth is really I have no evidence for why this problem is hard. The only evidence we have is that we didn't succeed in breaking it. There's no other evidence-- the fact that it can be broken-- it can be broken in polynomial time by quantum computers, yeah. And even classically, we have subexponential algorithms. So we have some nontrivial algorithms for doing it but not in polynomial time. The best algorithms are something like time 2 to the n to the third or something like that. But this is the problem.

So now let me go to the construction of our hash function. So I'm going to fix a group of prime order. For the construction, I'm going to assume that I have a group of prime order. I'm going to tell you how I find the group in prime order in a second, but think of a group that has a number of elements in the group is prime.

So let G be a group of prime order, which just means that the number of elements in G is prime. That's all it means. And now here's my-- and now, every element except for the unit now is a generator. So it's easy to find generators here. And now here's my hash function. My hash function depends on G and hash-- so what's the key?

OK, so we have a generator G. You can think of G as fixed. That's just a generator. If you want, you can choose it also randomly because any element is a generator here. H has a key, so-- OK, so now the hash function. So gen and 1 to the lambda. This is like sub lambda. What it does is it chooses random H and G, a random element. That's what it does. That's the key, OK? So hash key is just H, just the random element.

Eval, so eval takes as input-- before, we set eval takes as input hash key any x, any x. Now, no-- now less. So that's what I'm saying. Let me denote the number of elements in G by q. Now, eval takes as input a hash key and x. But now, x is not everything. It's just in zq times zq. So it's bounded. Later, we'll show how to get everything, but for now I'm bounded. It goes from zq to zq. And what it does, it generates-- so let me call it x0 and x1 because it has a pair. And what it does, it gives you G to the x0 times H to the x1 in the group, multiplication in the group. That's the hash function. So this is an element in G.

OK, this is the hash function. Now, why is it even shrinking? Because this is-- the group has only q elements. So it's like this is q elements, and this is 2Q elements. This is 1 out of q, and this is-- you have one out of-- you can think of G as embedded in like-- you can think of G as isomorphic to zq. So if you think of q, if you can think of q as like in 0, 1 to the lambda, then here you have 2, 0, 1, like 0 to the 2 lambda.

But let me actually make it more concrete for you. So it'll be just easier to think about it because I said a generator-- let me just prime-- let me just give you the concrete example you can think of. Look at zp star, so just multiplication mod p, where p is a safe prime. Safe prime I mean it's 2q plus 1 where q is prime. OK there's a lot of safe primes. So it's easy-- choose one at random. Check if it's safe. You can check primality. And if you choose enough times, you'll hit a safe one. There are enough of them to hit one with good probability.

And now, G is just going to be the set of quadratic residues, mod p. So this is all the x squared mod p for x in 1 up to p minus 1, all the squares. OK, how many squares are there? Like half. So zp star has p minus 1 elements, 1 up to p minus 1. And half of them are square. So the number of quadratic residues is p minus 1 over 2, which is exactly q.

So really, one can think if you want to be-- think of the quadratic residue as mod p. OK, that's-- so this-- great, so now you can think of this is all the quadratic residues. This is all the quadratic residues. And sorry, sorry, this is in zq. This is in zq. This is quadratic residues, mod p. And you can embed the quadratic residues mod p very easily. You can embed them in zq. You can represent each element here in zq.

How? Take any quadratic residue. Find the two square roots. So if you have x, it's a quadratic residue. So look at plus or minus square root of x. One of them is going to be smaller than q. One is going to be bigger, one's going to be in the upper half, one's going to be in the lower half. Write the one that's in the lower half. That's a way to express the quadratic residues as zq.

The reason I'm telling you that is because I want you to think that-- you can think of it as a hash function that goes from zq times zq to zq, OK, because you go from zq to zq to quadratic residues, mod p, and then just efficiently convert this to the square root. Which one? The one that's smaller, the one that's between 1 and q. And that will be the output. You can go from one to the other very efficiently. So this is a way to represent the quadratic residue. So it doesn't really matter, OK?

And the reason I'm representing the quadratic residue as an element in zq is just so you see that it's really shrinking, that you can see, OK, I have from zq to zq, from zq times zq to zq. That's my hash function. OK, the fact that it goes from zq times zq to zq, I'm literally going to use this fact to increase the domain to 0, 1 star.

But before I increase the domain, why is this a hash function? So in other words, what do I need from the hash function? Well, there's no opening here, so it's not a local opening. So this doesn't hold. But I want to make sure that it's collision-resistant. Why is it collision-resistant? So I want to argue that I cannot open if someone gives me-- if an adversary gave me a value, a hash value, he cannot open in two different ways.

So let me actually try to argue that you cannot open in two different ways. I'll do it here. So why? Suppose there is a cheating prover that can open in two different ways. Then I argue that I can break the discrete log problem. I can just use them to break the discrete log problem. Why? So let's say I get H. I'm an adversary. There's this G, and I got hash key. Sorry, I'm going to do adversary discrete log. So I got H, which is supposedly like G to the x. I'm going to find x.

Here's how I find x. I got H. I'm going to give this adversary here the-- so if I can find a collision, so suppose there exists a collision finder-- let me call him B-- then I construct A that uses B to break the discrete log. How do I break the discrete log? He gets H. He needs to find x. Yeah, he has G. He gets his input H. He needs to find the discrete log of H.

How does he find the discrete log? He goes to the collision finder. Let me write him as C for collision. That's better. He goes to the collision finder, and he tells me, oh, here's your key. Hash key equals H. Now find the collision. Now we find the collision. It's a V and two inputs. So I don't know. He gives a V and two openings, so two inputs.

So let me call one x0, x1 and other x0 prime, x1 prime, such that-- this is a preimage and this is a preimage. So g to the x0 H to the x1 is equal to G to the x0 prime H to the x1 prime. That's a collision. Now I claim, if he gives me this, I can use this to find the discrete log of H with respect to G. How? I just-- what is this? This is just G x0 minus x0 prime equals H x1 prime minus x1. I just moved arithmetic. And now, H is nothing but G to the x0 minus x0 prime divided by x1 prime minus x1. This is the discrete log. Done.

So what is the discrete-- he gives me a collision. What is the discrete log of H with respect to G? It's just x0 minus x0 prime divided by x1 prime minus x1. Now, where did I use the fact that the G-- remember, I said G needs to be a prime order group. Where did I use the fact that it's a prime order? Because this-- you compute this mod q, the order of the group. If it's not a prime order, it's not clear you can divide. Not every element has an inverse if you're not in a prime order group. So I want it to be in a prime order group so that every element has an inverse.

Now, another thing you need to make sure is that one can say, wait, x1 prime may be equal to x1. And then again you're out of luck. But if this-- if these are equal, then these have to be equal because if these are equal means this is 1. And the only case where G to the power is 1 if it's the power of 0 because it's a generator. Any other element is not 1.

So if they're different, if these two are different, that's what a collision means. Then x1 prime must be different from x1. And therefore, we can divide. And we're done. So this is the proof that this is a collision resistant. OK, any questions? Yeah.

AUDIENCE: Do you have any way to sample H during gen?

PROFESSOR: Do I have-- again?

AUDIENCE: A way to sample H during the generation of--

PROFESSOR: What do you-- what do you mean-- what do you mean to--

AUDIENCE: To just take H-- the element without necessarily learning something about.

PROFESSOR: So H is just, I choose it randomly from G. The key generation algorithm of the hash function, H is just a random element in the group. So if you use this specific group, you choose zp star, all the elements mod P, where P is a safe prime. You make sure P is a safe prime. And the way you choose h is a random quadratic residue. So I choose a random x, and x squared is my H.

So the key gen algorithm is very simple. And note, by the way, here you can indeed-- it's also random. There's no secret-- going back to the Kilian-Micali protocol, that the first message hash key is just random. It's random quadratic residue. There's no secret. You can send the randomness. There's no secrets in the hash key or anything like that. It's a very, very, very simple construction. This is, by the way, known as Pettersson Commitment for those who studied cryptography. It also has some hiding properties, which are nice, but we only care about collision resistance here. So we don't care about hiding. Any other questions about why it's collision resistant, about a construction?

So let me just say, by the way, as a side note, these kind of groups of prime order, in particular quadratic residues mod P, where P is a safe prime, is a very useful of group. We use it a lot in cryptography. So it comes up in many, many primitives. Working with groups of prime order is very comfortable. And the examples we have often is either quadratic residues mod P, where P is a prime order group, or elliptic curves, which is a whole different story. OK, so the next-- oh, yeah.

AUDIENCE: So all of [? this ?] prime order are isomorphic, right? They're all just cyclic?

PROFESSOR: Yeah.

AUDIENCE: So the special thing is in the actual structure of the group. It's like the representation of the elements or something. And the way that it's like-- yeah, you can't-- even though it is isomorphic to the cyclic group definition, you can't compute that [INAUDIBLE].

PROFESSOR: Exactly, exactly, exactly. Yeah, isomorphic is an information theoretic term. It's not clear that you can efficiently go from one to the other. Yeah, exactly, exactly. Good. Good point. Yeah. OK, so if there's no further questions, let's just go to show how to go-- how to get local-- how to go from collision-resistant to one with local opening.

So OK, this fell short in two ways. First, this construction goes only from zq time zq to zq. It doesn't go from 0, 1 star to zq. Second, there's no notion of local opening, which is also a problem. So what we're going to do now is we're going to show, actually, a generic transformation from any hash function that goes from zq times zq to zq that's collision resistant to the one takes to what we want.

So now we're going to do the construction from with local opening and arbitrary domain. And here is how it goes. So-- and this is from-- I'm going to assume I have-- which is gen and eval like above. But eval goes-- let's say-- I said zq times zq to zq.

Let me just write it as 0, 1 to the lambda times 0, 1 to the lambda to 0, 1 to the lambda. I'm writing it because that's how we usually write it, but if you want to think of zq, zq, zq, that's totally fine because that's what we saw. It's just that most constructions of hash, we think of it as 0, 1 to the lambda. That's why I write it this way. But everything I say you can replace 0, 1 to the lambda with zq, and it will work.

So I have that. I want to now get an entire family of hash functions from-- I want to make the domain much bigger, and I want to get local openings. So how do I do it? Actually, I do it in one shot, OK? I get the domain bigger and get local opening at the same time. And here is the idea.

So I do it via what's called Merkle Hash. It's an idea by Merkle. It's really beautiful, clever, and simple. And here's what he does. He says, look, you want to hash something. If it's less than 2 lambda bits long, done. You have a hash. Use your hash. But what if it's longer, if it's more than 2 lambda bits? So do it iteratively. What does that mean?

Take your input. Let's say-- OK, sorry. First gen, so let me-- the gen is exactly the same as gen. It's the same thing. Eval is-- so I need to give these algorithms, gen, eval, open, and verify. Gen is this gen. You have a gen for this algorithm. Use this. Now you have a hash key.

What about eval? Well, if your input is small, smaller than 2 lambda, you can use this eval algorithm. You're done. What if it's bigger? So if it's bigger, the idea is, if you have only-- so convert your input to blocks. If you have only two blocks or less, you're done. We have an eval algorithm for you. We're good. But what if you have, let's say, four blocks? What do you do? So the idea is oh, don't worry, apply eval to these two.

So you have a hash key. Compute eval with hash key and these two. You get a value. Compute eval with hash key, same hash key on these values. Now, I don't want to open-- I don't want to output both of them. That's 2 lambda. I want to get lambda. So no worries. Apply eval. And this is your value.

Now you're saying, wait, what if I have more? No worries. You have two more? Sure, apply eval. You have more? Apply eval. Now you can apply eval, and you can apply eval. So you can just apply eval, eval, eval, and you're done. That's really the algorithm. Now, there are two problems with the-- OK, I didn't tell the entire story for two reasons.

So first of all, there's an issue of how many blocks are there really? So usually, what we do, we say let's assume that the number of blocks is 2 to the L for some L just to make this tree complete. So usually, what we say, we don't actually do x from 0, 1 star. We assume that all the inputs are of size 2 to the L for some L. Now you're saying wait, but that's not the case. Oh, with padding, you can make it the case.

So the idea is, given x, always put-- so you want to do eval of x. Don't do eval of x. Put eval x. Append with 1 and 0's. How many zeros is the minimum amount that will give you size 2 to the alpha for some L? It increased the length of it, yes, but only by a factor of 2. So it's not that bad. So now--

AUDIENCE: [INAUDIBLE] 1?

PROFESSOR: OK, good. Great. Like, why am I putting here 1?

AUDIENCE: Otherwise, you don't know [INAUDIBLE].

PROFESSOR: Exactly. Otherwise, you don't know. Is this part of x or [INAUDIBLE]? So this-- so now when you have padding you know all this is padding until the one including, and this is the x. So this is just to make it 1 to 1. Great question. Yes.

AUDIENCE: Does it cause issues if you take this tree and you pad it this way and you were very close but only barely above a power of 2? The entire right-hand side will be a ton of hashed 0's.

PROFESSOR: Yeah, you're right. So here's the thing. If this is exactly power of 2, exactly, I'm always going to add a 1 and append is-- so I shouldn't have added a 1. But my algorithm is always append a 1, and make it a power of 2. So if it was originally a power of 2, unfortunately, I added 1. Now I'm stuck. I need to add a bunch of 0's. So all these are going to be 0's. Yeah, you hash that.

AUDIENCE: OK, I guess it just seemed-- OK, never mind. Maybe the rest of the instruction will make it clear.

PROFESSOR: OK, so there's another part of the construction that's missing, and I'll tell you. But there was another question, or?

AUDIENCE: I was just wondering if now this will not impact the correctness.

PROFESSOR: OK, good. So the question was, will it impact the correctness? And the answer is it won't impact the correctness if the eval algorithm now-- so let me be clear on what the eval algorithm is. The eval algorithm takes as input x. It adds a 1 and is the minimum amount of 0's to get a power of 2. Once he has x that's the length is a power of 2, like 2 to the L, then he does eval, eval, eval, eval, eval, eval, eval in a tree-wise manner. And now he's just going to have one root.

The reason I made it because it's not a power of 2, then there's kind of dangling roots hanging out. And I don't want to deal with these dangling roots. So I'm just going to make it a power of 2. Now, what is the output? The output is the last hash value. Usually, we denote it by root because it's the root of the tree. So often, if you see hash you'll see-- they denote the hash instead of V for value. They denote it by root.

But the output is not only root. And I think that goes back to maybe your concern. The output is not only root. It's also the depth of the tree. So the hash value, so eval given hash key and x, what it does is, one, compute, let me call it x prime, which is x1, 0 like the padded. This is of length 2 to the L for some L. Then 2 compute root via tree. So you compute all these hash value, hash value, hash value, hash value, hash-- compute all these hash until you get to the root.

And output root and depth of the tree. This is very important. You have to output the depth. If you don't output the depth, then it's easy to find collisions. The reason it's easy to find collisions is I can, let's say, generate this randomly. Compute, compute. And then I can either open to these two and say, oh, I just hashed to these. Or say I hash to this. And it's a collision. Both of them hash to the same value. They're different sizes.

So without the depth, one can argue you cannot find collisions of the same size, of the same length. But it is easy to find collisions for inputs of differing lengths by just the attack that I said. I'm an adversary. I'm going to choose a random long input. Honestly compute the root. And then open either to this layer, say, or open to this layer, open to this layer. All of them are going to be kind of valid openings.

So I found collisions. Once I have the depth, I can't do that anymore because it won't correspond if I open this, well, it doesn't correspond to this depth. This is a depth 1, and I output a depth-- yeah.

AUDIENCE: The problem is that this is no longer lambda length [INAUDIBLE].

PROFESSOR: Good, OK, OK, very good. What you're saying, yeah, you lied because this root is of size lambda. This depth can be actually even bigger than lambda. Yeah, you promised lambda. OK, so let me tell you how I deal with it.

If you're-- OK, this is a nitpicky. I can make it lambda. How do I make it lambda? You know what? I'm going to run-- here's my new gen algorithm. The gen algorithm on input 1 to the lambda will run the original gen algorithm with lambda over 2. And then, this is of size lambda over 2. A root now is of size lambda over 2.

So now I have lambda over 2 t of, I have lambda over 2 spare to play, and I can use this for the depth. But actually, you know what? Let me make it simpler. You know what? If you'll excuse me, I'll write V to be 2 lambda. Make it 2 lambda. Let me allow myself here slack.

AUDIENCE: Can the depth be unbounded, though?

PROFESSOR: Good, you're right. You're saying, wait, but the depth can be unbounded. So you know what? If it's more than 2 to the lambda-- sorry, if the depth is more than lambda, I will output fail. The point is-- OK, so-- OK.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Let's do 2 to the lambda here. I always think of everything. The inputs are always at most size 2 to the lambda. That's the truth. So just think of the input. We also have correctness only for inputs of size 2 to the lambda. Our guarantees hold only for inputs of size 2 to the lambda. So let's just think of it as size 2 to the lambda. You're right. If it's not 2 to the lambda, I don't have any guarantees. So why even talk about more than 2 to the lambda?

So I guess the way to think about it, you can handle any input length, but the security parameter-- the hash function will grow with log the input length. So you're right. You're right. If you want to use any star, you should-- let's think of that always bounded by 2 to the lambda. Anyway, we'll need that for completeness. We'll need it for soundness. We need it for everything. So just assume that it's 2 to the lambda.

AUDIENCE: I think we can also mark each node with a bit [INAUDIBLE] delete or not delete?

PROFESSOR: Uh-huh.

AUDIENCE: [INAUDIBLE] lambda there and lambda plus 1 [INAUDIBLE]?

PROFESSOR: So you're saying you can mark each node, but for what? What's your purpose for--

AUDIENCE: So we mark [INAUDIBLE] with an extra bit. That's 1. And mark [INAUDIBLE] that are not [INAUDIBLE] an extra [INAUDIBLE] 0.

PROFESSOR: 0, OK.

AUDIENCE: And then I think that's also [INAUDIBLE]?

PROFESSOR: Oh, you're saying there's an attack.

AUDIENCE: No, I'm saying that it solves it.

PROFESSOR: Oh, solves, solves which problem?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Oh, you're saying-- OK, OK, yeah, what you're saying is maybe we can solve-- OK, so you're saying maybe we don't need to give the depth here. And instead, what we can do is you're saying the honest truth, the honest person, he will add 0 to everything here. And he will add a 1 to everything here.

AUDIENCE: [INAUDIBLE] lambda attack as opposed to just a single bit to avoid getting unlucky with hash [INAUDIBLE].

PROFESSOR: OK, yeah, so you need to make sure-- one bit is probably not good because-- I guess my concern is if you're not-- this is what I'm going to do if I'm honest. But I'm not honest. So I'm going to go ahead with my original crazy thing of-- OK, fine. If you want, I'll put here a 0, and I won't put 1. I'm just going to compute without your tags because I'm malicious. And now, when I open, maybe some of them are going to be 1, and I'm lucky. You see what I mean? Or 0, and I'm lucky.

AUDIENCE: But if you add different tags for that, you add tags for 0 on the very bottom row, and you also add tags of 1.

PROFESSOR: Yeah, but you're right. OK, but now I don't do. That's what I should do, but I'm malicious. I don't do that.

AUDIENCE: So what two inputs would you give that give the same hash?

PROFESSOR: So if I'm allowed to put 1 or not put 1, I'm always going to put 0. Oh, OK, sorry, what you're saying-- OK, OK, now I understand. You're saying if I want to open it, this has to be a 1.

AUDIENCE: Yeah.

PROFESSOR: And therefore--

AUDIENCE: [INAUDIBLE]

PROFESSOR: Now I understand what you're saying. OK, got it. So you're saying this may solve the problem. Ehm, it's a good suggestion. You're like yeah. Eh.

AUDIENCE: [? Now, ?] I have a question.

PROFESSOR: Yeah, sorry. Yes, Ted.

AUDIENCE: [INAUDIBLE] needing to increase the length of the block at each level by 1 [INAUDIBLE]--

PROFESSOR: Yes.

AUDIENCE: --which would get the exact same thing?

PROFESSOR: The opening will be-- the opening will grow with the depth.

AUDIENCE: Well, if we're-- we need to have a hash function which will allow us to walk in the-- this is not a [? leaf ?] intermediate thing. So the hash function needs to now take like lambda plus 1 bits in at number 1 and then plus 2 and bits at level 2, et cetera.

PROFESSOR: Oh, you're saying it will grow. I see.

AUDIENCE: Presumably, the hash function is still like k to k goes to k. But in order to keep adding additional padding--

PROFESSOR: So you're saying if you pad here, this will go to k plus 1. It'll go to lambda plus 1.

AUDIENCE: [INAUDIBLE]

PROFESSOR: And then you add 1, it'll go to lambda plus 2.

AUDIENCE: Yeah, so you've just got to run the plus [INAUDIBLE].

AUDIENCE: Can't you--

PROFESSOR: I wonder if you can get around this, though.

AUDIENCE: I don't have to power of 2, don't make it a complete tree. You make it like [INAUDIBLE].

PROFESSOR: It's a good question if you could do it with completeness-- without blowing up full completeness.

AUDIENCE: It seems [INAUDIBLE] like a [INAUDIBLE] domain extension has Merkle-Damgard construction that can be [? done ?] with 2 lambda plus 2 bits. You can still shrink it, shrink that into a 2 lambda bit. And then do the hash [INAUDIBLE].

PROFESSOR: Yeah, yeah, yeah, you can probably-- that's what I was thinking. You can go-- you write that it grows, but you can maybe shrink a little bit and then continue, and shrink a little bit and then continue.

AUDIENCE: But if you make those [INAUDIBLE] not have to open.

PROFESSOR: So we didn't talk about opening yet. Look, you gave a really interesting idea that I'm trying to think if I can make it into something in the P set.

[LAUGHTER]

OK, I'll think about it offline. Anyway, we have five minutes, so let me quickly talk about how do you do local opening and about collision resistance of this thing. So first, we managed to do domain extension, OK? So now the domain can be up to 2 to the lambda. And the hash value is just lambda plus lambda because we assume that the inputs are 2 to the lambda. The depth is at most lambda. Yeah? So we're happy with the size.

How do you open? So let's say I want to open to one point here. I want to open to this bit. How do I open to this bit only? Well, what I do is I identify the block that it sits on. So now I know what I want to open is this-- I'm going to open to this entire block. It's only lambda bit. So remember? The opening is poly lambda bits. I can allow poly lambda bits in my opening. So I'm going to just open to the entire block. So I can give you the block, but how do you know it's the right block?

So what I do, I give you-- you know what? I'm going to give you the block and the sibling block that is needed to compute the parent. So I'm going to give you both of these. Now you can compute the parent. And then I'm going to give you the sibling block that is needed to compute the parent. And then I'm going to give you the sibling block that is needed to compute the parent.

So for every layer, I give you the sibling block that you need to compute the parent. So the number of blocks I give you is like the depth. At each layer, I give you a sibling block. So its depth number of blocks. Each block is the size lambda.

So I give you polynomial, lambda squared bits, OK? So this is the opening. The opening just contains the block of the bit you want to open with the sibling blocks of each layer, the sibling blocks of the parent, the sibling corresponding to the parent all the way up.

How do you verify it? Well-- so that's the opening. So the opening are all the kind of green-- well, this green you can compute on your own. So you don't need it, but think of it all the green blocks here. How do you verify? You just verify. You check. You check that these two hash to this. These two hash to this. These two hash to the root, the route of the-- that's the hash value.

So actually, this you don't need to give because you can compute. This you don't need to give because you can compute. But this you need to give. And the point is you can ask the verifier to recompute this, recompute this. But at the end he needs to make sure you're consistent with the root, with the hash value that he committed to.

AUDIENCE: So you're saying you just needs a siblings.

PROFESSOR: You just need the siblings, but it doesn't matter. Yeah, you can just give one sibling. You can give d instead of 2d. That's the opening. That's the verification. Yeah.

AUDIENCE: How do you ensure this opening doesn't like-- it's consistent? It doesn't-- say one sibling block is different. The next time you say open on a different block.

PROFESSOR: So OK, there's a-- so the only thing I need to check is his collision-resistant. I don't know. What do you mean by consistent?

AUDIENCE: Give a different sibling block to the next [INAUDIBLE].

PROFESSOR: Fine, I can do whatever I want. Look, I can do whatever I want.

AUDIENCE: [INAUDIBLE] block--

PROFESSOR: I can do whatever. Look, I'm a cheater. I'll do whatever I want, whatever I can-- whatever I can do and get away with, that's what I'm going to do. The question is, I think what the honest prover should do. The honest open algorithm, he should give all the correct blocks. Now you're saying, he won't. What if he doesn't?

So now I'm going to tell-- here's what I want to argue. Look, he may not. I don't know what he's giving, OK? All I need to argue is collision resistance. First, note completeness-- correctness, as I said, if you're honest, you're going to be accepted. If you're honest, you generate this gigantic thing like you should have. You have the entire tree. And when you want to open this, you actually give all these. Everything should hash. The hash is good. It computes correctly. And you'll accept with probability 1, OK? Always. That's not a problem.

What your concern is like, wait, OK, I tell the cheating person who wants to find collisions, this is what you should do. Give the opening. You're saying, he won't give this block. He will give this block. Or who knows what he's going to do? Who says he can give whatever he wants? So how do you know that this is-- how do you know that this is collision-resistant? You're not happy.

AUDIENCE: [INAUDIBLE] because probably [INAUDIBLE].

PROFESSOR: Yeah.

AUDIENCE: Yeah, I was more concerned about the soundness, the soundness that we get from there. How do we know he can't-- each time you query, you just kind of like-- for his proofs of those sibling blocks, you can just construct them in such a way that it ends up at your root. How do you make sure that you can't just construct them so that in [INAUDIBLE]?

PROFESSOR: You should be concerned. So first, when you say soundness, you mean collision-resistant. I wrote soundness, but that's the-- yeah, that's kind of thing. Yeah, and you're right. You're saying, how do you know? He does whatever he wants. How do you know that at the end of the day, somehow he doesn't manage-- maybe he manages to mix, max, and do it-- and magically get to the root in two different ways?

So the answer is he can't, actually. The only way he can generate two different openings for the root, the only way-- if he could generate two different openings for this, one block and a different block with the valid openings, I argue I can find collisions to the original hash function that I started with from 2 lambda to lambda. The reason why I find collisions-- we're out of time, so we'll have to do it next week. So Friday at same time next week. Finally, we have consecutive weeks.