[SQUEAKING]

[RUSTLING]

[CLICKING]

PROFESSOR: So our last class, which is happy, because we get to go on break, but there's so much more I want to tell you guys. But that's all we have time for. There are cookies here to make our last class sweet. So please come and get some if you want. They're very small. So you can take two. If you didn't want to commit to a large cookie, you don't need to.

So today, finally, we're going to actually construct the SNARGs that we've been after. That was kind of the goal is to construct SNARGs, Succinct Non-Interactive Arguments, and like this entire course was a buildup to construct these SNARGs. And today we're going to see how to construct them.

There are cookies, guys-- cookies, if you want, from flour-- really good ones. So feel free to take some.

So what we're going to do today is show how to construct these SNARGs. And we're going to-- last class, Zhengzhong showed you his really beautiful work and constructing BARGs, which are Batch Argument. And on their own, you can say, why do we care about batch argument? I'm going to recall the definition.

But I mean, it's interesting on its own. But what makes it super interesting is that we can use it in a pretty straightforward manner to construct SNARGs. So that's what we're going to see today. We're going to see how to use these BARGs, that I'm going to recall the definition in a second, and also how to use-- I think Zhengzhong called it somewhere statistically binding hash functions. It's also known in the literature as somewhere extractable hash functions. Both are used. So I'm just going to use the other, just so you're familiar with both formulations, both names.

So what we're going to see today is how to use these BARGs with these hash functions to get SNARGs. That's kind of the plan for today. So let me first recall what a BARG is really fast. So a BARG, you want to prove that many instances belong to an NP language.

So a BARG for an NP language L consists of three algorithms. Gen takes as input a security parameter, the number of instances you want to batch. Maybe I'll close-- thank you. Can you close the door? Thanks. k, which is the number of instances you want to batch, and n, which is the length of each instance. And then output some CRS.

Note lambda is given in unary because we want them to run in time poly lambda. But k and n are given in binary. So because it's a polynomial time algorithm, it runs in time polylog in k and polylog in n, the instant size.

And then the prover algorithm takes as input the CRS, and then x1 up to xn-- xk-- sorry. So k elements with witnesses and outputs, a proof pi-- but importantly, this proof should be small. So this pi is of size poly log-- or poly n security parameter, but log k. That's important.

We want it to depend only polylogarithmically on k. That's kind of our goal. And then, maybe ideally, also log n. But the point is we want it to be succinct. Otherwise, it's trivial. You just output the witnesses.

And the verifier takes as input CRS, the instances, and the proof. And it outputs 0, 1-- accept or reject. And this, when I say the size of the proof is this, actually, you can make it bigger or smaller, but we want it to be succinct.

Sorry. Sorry. I didn't-- we want it to depend only logarithmically in k and so on, but we allow it to depend polynomially in one witness. So I can put it here, a single witness, one, not many.

I really want to be as succinct as possible, but what we can get is a BARG that depends-- grows with one instance, one witness, as opposed to k instances. So this is kind of the algorithms, then we want completeness and soundness. Completeness is the usual thing that just says, if P has valid witnesses and it generates a BARG, this BARG is going to be accepting with probability 1 over the CRS generation. So that's the standard completeness.

And then the soundness property that we have, ideally, you would want to say that-- so in soundness, computational soundness, we have adaptive soundness and non-adaptive soundness. And we want to say that a cheating prover cannot convince you of a false statement. So he cannot generate-- he cannot give you x's that are not all in the language with a valid proof. That's soundness.

There's an issue with it. When does the cheating prover get to choose these x's? Is it before he sees the CRS or after he sees the CRS? If he needs to decide that before he sees the CRS, then it's not adaptive. If you get to decide it after, then it's adaptive.

Ideally, we want to say it's a cure for all very-- as big class provers as possible. And thus we would like to have something against adaptive provers to say, even if a cheating prover chooses this x1 of xk, adaptively, after seeing the CRS, if one of them is false, he cannot generate a proof. That's what we would like to say.

What we can say is what's called a semi-adaptive, something in between. So let me just write it here. So semi-adaptive soundness just says the adversary needs to tell you which-- so he's going to give you k instances. He can choose them adaptively, but he needs to choose the i on which he's going to cheat. So he's a cheater.

So one of the xi's is not in the language. Which i, which location i, the xi, is going to be not in the language? That should not depend on the CRS. So it says that for any poly size P star and for every i, which, of course, may depend on lambda, because for any lambda you have an i, because the i is between 1 and k. And k grows with-- can grow with lambda.

So for any i, you want to say the probability that P star, given the CRS output x1 up to xk and pi, such that xi is not in the language and pi is accepted is negligible. So xi, one of them is-- the xi one is not in the language. And the i was chosen before he saw the CRS.

So for any P star, he tells you, I'm going to cheat on location i. Now, he's given a CRS. The probability that he gives you an accepting proof for x1 of xk that he chose adaptively after seeing the CRS, the probability that it's accepting-- so CRS x1 up to xk and pi is 1. And x is not in the language. xi is negligible. This equals a negligible.

So again, for any cheating prover, he needs ahead of time, I'm going to choose my xi and xk adaptively after I see the CRS. But I'm going to cheat on location i. The probability that he succeeds in cheating in location i in the accepting way is negligible. That's the semi-adaptive soundness.

And last class, Zhengzhong kind of showed you the high level of how you construct such a-- like the construction of such a BARG. Any questions? Yes.

AUDIENCE: Is this required? The indices other than i, are those required to be in the language?

PROFESSOR: No, we don't care. So yeah, good question. So the P star can choose the rest of the x's however he wants-- in the language, outside the language-- however he wants. But he wins if the xi is not in the language and the pi is accepting. And he wins with negligible probability. Yes.

AUDIENCE: Does the-- transforming it into fully adaptive, would it imply guessing?

PROFESSOR: Yeah. OK, good. So the question is, can we transform this into a fully adaptive SNARG? So converting it into a fully adaptive SNARG is non-trivial. And I'll tell you why. You can guess i. You can guess i. And you'll guess it correctly with probability 1/k in general.

But here is-- so you can do it. But here is the issue. I'll tell you what the complication is. In the analysis-- so I don't want to recall too much of the construction, but the high level idea of how these BARGs are analyzed, even without knowing the construction, the way they're analyzed-- in the CRS, so in the proof, in the analysis, we say, suppose there exists some cheating prover that cheats on some i.

In the CRS, I'm going to change the CRS now. I'm going to tell the cheating prover, here's a new CRS. And in this new CRS, I'm going to hard wire i. I'm going to put some information about i in a way that's unnoticeable.

And when we'll see, I'm going to call the somewhere extractable hash, or what Zhengzhong called SSB hash, somewhere statistically binding. And then I'll remind you a little bit how we-- where we stick the index i. But in the analysis, we say, let me change the CRS to a different CRS. It has a very different distribution, but it's indistinguishable. It looks like the original CRS.

So P star can't distinguish between the two. So he'll still give me a valid BARG. And now I'm going to say, in this new CRS, actually, there's no way he can cheat on location i. There's no way he can cheat.

What is the issue? So now you're saying, why do we need-- why can't we do fully adaptive? The issue is it may be the case that once we switch the CRS, it switches where he's cheating. So here's an alternative way. Let's say the prover tells you, you know what? I'm going to cheat in location i. But I can be adaptive-- chosen based on the CRS.

So now in the analysis, I'm going to say-- I'm going to guess where he cheated. I'm going to say, let's guess he cheated in location 7. I'm going to hardwire kind of the CRS to depend on this location. And then the cheating prover doesn't cheat on 7. It cheats on something else. He can kind of evade me.

Now, you're saying, oh, if he cheated on something else, I can tell the difference. So I shouldn't be able to tell. But the thing is you don't know what's in the language, what's not in the language. So the problem is whether xi is in the language or not in the language can be very hard to tell. It may take-- and that's why kind of getting adaptive-- just guessing is not good enough, because he needs to-- the issue is if you could tell efficiently whether x is in the language or not in the language, then, yes. But because it can be very hard, that's where things become tricky. So that's-- so you can guess, but it's not good enough is the answer. Yeah.

AUDIENCE: Even if he commits--

PROFESSOR: Huh?

AUDIENCE: Even if he commits to doing it, [INAUDIBLE]?

PROFESSOR: Well, he can-- what do you mean he commits? So you're saying let's change the scheme kind of and have him add commitments?

AUDIENCE: Well, I guess it might be different.

PROFESSOR: I guess the issue is-- look, when he's cheating, we don't know whether-- he gives you x1 up to xk and a proof pi. Now, he cheats if one of them is not in the language. That's when he cheats. It's hard to know whether one of them is in the language or not in the language.

And the concern is he can say i. He can say whatever he wants. He's going to cheat. So nothing he says we believe. And the concern is that maybe when we switch the CRS to depend on i, also the xi becomes in the language. And then what do you do? And now you can say you can't tell the difference, because xi in the language and outside the language, they look the same. So that's why guessing is, in some sense, not good enough. It's not helpful. So this is-- any other questions?

So this is one ingredient that we're going to use to construct our SNARG. The other ingredient is a somewhere extractable hash, or SSB hash. So let me write it out. I'm going to write it here, so we'll keep it on the board.

So let me just recall the definition. Somewhere extractable hash family consists of many several PPT algorithms. Let me start writing. So the first one is gen. So now we're constructing a hash. So gen takes a security parameter, the length, how many bits we're going to hash. So let's say n bits we're going to hash.

And how many-- so let's say in the location i, this is going to be i and n. This is where we're going to be binding on, or where we want to be extractable on. And it outputs a hash key and a trapdoor. This trapdoor is going to be-- we're going to use it to extract from the hash the i-th location of what we hashed. So there's a hash key and a trapdoor.

I know when Zhengzhong taught it, then he just said, hash key, and he put kind of the trapdoor at the end. So I'm going to write it with a trapdoor. So gen outputs a hash key and a trapdoor. And then you have eval algorithm. It just takes the hash key, an input x and 0, 1 to the n, and then outputs a hash value v.

And then you have open, where you take a hash key, the x, and some index j that you want to open to. And it outputs an opening rho or rho J. And this should be in 0, 1 to the lambda, or poly lambda. Similarly here, this should be small-- 0, 1 to the lambda or poly lambda.

And then there's ver that verifies. So ver takes a hash key, a value. And now let's say you want to say, I'm going to open location j. Here's my b that opens. And here's the opening. And he checks. Is it a valid opening or not? So it outputs 0 or 1.

And finally, so it consists of five PPT algorithms. So first, you generate a hash key with a trapdoor. Then there's an eval algorithm that given an n-bit string outputs the hash value v-- short, succinct. Then there's open that you can open, given that you can generate an opening for a specific index j. So you generate row j, which is an opening, and anybody can verify the opening using only the succinct hash value. So given a hash key, the succinct hash value, the index, the bit, you can verify the opening.

And finally, that's the extract algorithm. And it takes the trapdoor and the value, and it outputs a bit. And this bit should be xi. So let me say, the properties-- so the first property is completeness. That's kind of usually the straightforward property, which, essentially, says, look, if everybody's honest, you get what you expect, which means, one, the verifier would accept the opening, and b, the extract will output the i-th bit.

So it says that the probability-- so for every x, for every lambda, for every n, which is at most 2 to the lambda, we always restrict the length of what we're hashing to be at most 2 to the lambda, and for every i, we say that the probability that an open-- sorry-- that ver a hash key generated from gen, which is the eval, for every j. j rho generated by open. It always outputs 1. It outputs 1, probability 1.

And extract-- I'll just denote by-- extract, let me denote by just x for simplicity, given trapdoor and the output xi. And this holds with probability 1, where the probability is over what? Hash key and trapdoor are in gen. So essentially-- I won't write it. But the hash key is generated by this. v is generated by this eval. Rho-- or I'll call rho j-- is generated by open.

And this just says if everything was done correctly, the var always accepts, and the extract indeed will extract the correct bit. So this is just if everybody follows the protocol, everything should be as you expect. The soundness or binding, or the statistical binding, I should say, somewhere-- or I can call the FSB somewhere statistically binding condition says that on location--

Oh, before I go to FSB, there's another property, which is index hiding. Index hiding means that the hash key hides the index. If you see just hash key, you have no idea what i is. So it just says that for every i and j, if you look at hash key generated by-- I don't know-- call it i, or hash key generated by j, they're indistinguishable.

This is just a notation to say hash key generated where here I have i. And this is just a notation saying this is a hash key generated by giving j to gen. So the hash key hides the index for any i and j, the hash key generated with i, the hash key generated with j, they look the same. So that's the index hiding.

And finally is the binding condition. And the binding condition says that-- so the SSB, the Somewhere Statistical Binding, says that for the location i, that we're binding on, even an all powerful adversary that is given hash key cannot generate v and two different openings to location i, an opening to 0, an opening to 1 on location i.

So on location i, you're really [INAUDIBLE] theoretically bounded. So again, even in an all powerful adversary that is given the hash key with respect to i, a location i cannot open both to 0 and to 1, except with negligible probability.

So I hope to put everything on that, but I'll add one here. So I'll put here the binding condition, so the SSB binding, the somewhere statistically binding says that for every all powerful adversary a, the probability that a and hash key, where hash key is generated from gen 1 lambda n and i, the probability that he outputs any value v and rho 0 and rho 1 such that they would accept both, such that for every b, both 0 and 1 there on hash key vi b rho b outputs 1, this is negligible.

So on index i, the one that we're-- the hash came from that we're binding on, he cannot open to both 0 and open to 1 in an accepting way with accept negligible probability. Yeah.

AUDIENCE: Does v have to be generated from eval here or maybe [INAUDIBLE]?

PROFESSOR: No, no. Good, good. Great question. v is malicious. There is no-- a can be-- so the adversary, he's not following any guidelines. He's completely malicious. He generates v according to his own will, generating rho 0, rho 1 however he wants. And he's all powerful. He still cannot generate two accepting openings for location i. Yeah.

AUDIENCE: I'm just going to follow-up on that. That's the only reason that we don't get this for free from extraction.

PROFESSOR: Exactly. Good, good, good, very good. So let me just repeat what [INAUDIBLE] said. Look, in extraction, we said that we're binding, completely binding, because guess what-- from v, we can actually-- we can actually learn the bit xi. So xi is sitting there.

So yeah, if you're honest xi is sitting there. But if you're malicious, you can do whatever you want. And maybe for a malicious v, maybe there's no-- you can't extract anything from it. Yeah.

AUDIENCE: Is it the same as saying the probability there exists v [INAUDIBLE]?

PROFESSOR: Right. Good, good, good, very good. Yes, yes, yes. This is the same as saying that there exists a v rho 0 for any hash-- for almost all hash keys, the probability that there exists v rho 0 rho 1 is 0. So they don't exist. For almost all hash keys, this triplet that satisfies this does not exist. Yes. Yeah. Very good. Any other questions?

So the plan for today is to show how to use these two building blocks to get a SNARG. But before I do that, I want to say a few more-- first, I want to give you time to digest and to ask questions. Also, I want to say a few more things about these two primitives. Any questions before I-- OK.

So one thing I want to mention is even though the binding condition was only an i, I said only on this specific i, I promise you, you can't open two different ways. What about the other i? What about j? j can address to open two different ways? And the answer is no. Actually, he can't. And we don't need to state that explicitly.

From the FSB and the index hiding, I want to argue already we can conclude that a poly size adversary cannot open a poly size-- not all powerful, but a poly size adversary cannot generate v any index j, and two openings that will be accepted, except non-zero probability.

So in other words, I'm saying binding holds-- this local binding holds for every j. Not against all powerful, but for any PPT or poly size A, we get this binding condition for every j. Why?

AUDIENCE: Is it because of the index hiding property where you cannot distinguish [INAUDIBLE] index is higher or something else then?

PROFESSOR: Precisely. So the reason is this A, he doesn't know if hash key came from index i or index j. They're indistinguishable. So now if he can generate it for index j, then he should be able to generate it also for index-- so let's replace this with index j. He can't distinguish. So he'll follow the same j again at some point. And there'll be a contradiction.

So because he has no idea what's sitting here, the fact that he cannot do it for that index i, even information theoretically means he can't do it for any other index. Yeah.

AUDIENCE: So this index had to get really [INAUDIBLE].

PROFESSOR: No, this is [INAUDIBLE]. It's computational, actually.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Oh, good, good, good. So the corollary is, for any poly size, the corollary says that only any poly size A, the probability that A on hash key, let's say from i, same hash key, outputs v or v, j rho 0 and rho 1, such that the same thing here holds. Namely, for every b he generates accepting answers for both 0 and 1 is negligible.

Because if there exists some j that he does that for non-negligible probability, then let's replace hk with hash key generating that's binding on this, that's actually statistically binding on this j. A can't distinguish. So we'll still do this for that j with some non-negligible probability. And that he cannot do with that information theoretically. So that's why we get, like-- because index hiding, actually, just getting statistically binding on one of them is good enough. It will imply computational binding on all the rest.

Another thing I want to mention-- and this we'll use, actually, in our construction today-- I said that the hash is binding on one location, on xi. We can get it binding, actually, on l locations, not just one. How do I get it binding on l locations? I'm going to just generate hk1, hk2, up to hkl, where the first l is binding, and the first-- I'm just going to generate l hashes.

I'm going to ask the-- generate the eval on all these hashes. So the hash value now is going to be v1 up to vl, where the honest eval is going to take the x you want to hash and do eval hk1 of x, eval hk2 of x. So just hash the same x with all l keys. So now the hash value grows by a factor of l.

And then when you open a certain bit, a certain location, you'll generate all the openings. So just do the same thing kind of l times, just repeat it l times. And now what do we have from that? If you-- by repeating l times, so let me write it.

Let me erase the corollary. So I'll have room. So note, remark, by repeating l times, we get a somewhere extractable hash that's extractable on l locations. And moreover, another property that we're going to need-- even if I, let's say, gave you the trapdoor corresponding to some of the l-- not all the l locations-- let's say I give you trapdoor 1, trapdoor 2, and trapdoor 3, still, what's the location hidden in hk4? You have no idea what it is. So even if I reveal some of the trapdoors, the other indices remain hidden. And we're going to use this property.

So again, if we repeat l times, we're binding on l locations. But even if I dest-- I kind of reveal the trapdoor corresponding to-- I don't know-- a location to a couple locations, the rest of the-- so of course, if I give you the trapdoor, then all bets are off. You learn everything. You learn the i. You learn everything. And the trapdoor or trapdoors are such that once I give you the trapdoor, you learn the index. You learn everything.

But the other indices that do not correspond to the trapdoors that I gave are still completely hidden, because I did them completely independent. And this is something we're going to use when we're going to construct our SNARGs. So that's one thing we're going to use.

I want to mention a little-- I want to talk a little bit about the construction, because I think maybe in Zhengzhong lecture last time, this was missed a little, because he said it, I think, a little too fast. So I want to say a few words about the construction.

But before I talk about the construction, I want to say one more property about the BARGs that we're going to also use. So what we're going to use about the BARG, so we said that these gen P and V are polynomial time algorithm, probabilistic polynomial time algorithms. In particular, V, I want to talk about V. V is a polynomial time algorithm.

But V's input is really large. It has x1 up to xk. So polynomial time means it runs at least in time poly k. It needs to read its input. However, it turns out that the BARGs we have-- this is what Zhengzhong called a BARG for index languages-- if there exists a succinct description of these x1 up to xk, the verifier only needs to run in time that depends on the description.

In other words, we can replace-- so if-- I didn't bring my-- I forgot to bring my colorful chalk. Sorry. But at least you have cookies. I mean, what would you prefer, colorful chalks or cookies?

So if you can generate these x1 up to xk, if there's kind of a succinct description, then the verify runs in polynomial time in the description. So if, let's say, there exists some machine M such that for every i, xi is just kind of the Turing machine on input i.

So this kind of machine, you should think this algorithm, you should think it has kind of a succinct description of all the x1 up to xk. And you give him i. It always gives you the xi. If this is the case, then V the runtime, or I say time of V, it grows with the size of m as opposed to k. So it grows with m and does not grow with k.

So of course, if you give him x1 up to xk, he needs to read his input. His runtime grows with k. But if you can generate in a succinct way, then he doesn't need to grow with k. So you should-- another way to think about it, you can think of xi as m comma i. And the instance just says if you run m on i, xi, then xi is in the language. And that's why it's called index language, because it's like there's an index. Yeah.

AUDIENCE: So this means that there's still a log k dependence in V's runtime, right?

PROFESSOR: Yeah, yeah, yeah. There is a log k dependence, 100%. Sorry. Yeah, yeah. It just means-- yeah, exactly. That runs in time log k. Actually, yeah, you have i here, which is log k. So there is a log k dependence. Yeah.

AUDIENCE: [INAUDIBLE] the CRS is sublinear in k as well. Is that true?

PROFESSOR: The length of the CRS in general, you want it to be sublinear in k. In the SNARG construction, it's poly log k. Yes. But when we'll talk about SNARGs-- so in the BARG construction, we have the CRS. Most of our construction in the CRS is of size poly log k. The one under LWE that Zhengzhong showed is of size poly log k.

AUDIENCE: So I know that if we have a BARG that satisfies these properties and is a more tractable hash function, I guess you can always kind of add the property that you want here of being able to extract the witness from the BARG group itself. Is there a world-- like, do we have any BARGs without the assumptions that we need for somewhere extractable hash functions?

PROFESSOR: You're asking, is there a BARG for which we don't know how to extract.

AUDIENCE: Yes.

PROFESSOR: Is that what you're asking? OK, OK. So let me-- you're asking a great question. Let me kind of-- so when we said-- the soundness condition just says that a cheating prover cannot cheat-- cannot generate x1 up to xk, that one of them, the i-th is not in the language, and yet convince. Oftentime, also in our SNARG, in our SNARG construction, we want something stronger. What we want is that I can actually extract a witness from this cheating prover.

So I want to say this cheating prover, I can't extract all the witnesses. But for some i, I can generate a CRS that looks like-- just standard CRS-- but I can change the distribution of a CRS to be such that using some trapdoor about the CRS, I can actually extract from P star a witness for xi. And once I extract a witness, of course, it has to be in the language, xi. I argue not only it's in the language. I can even find-- get a witness from him. So that's called somewhere-- this is a somewhere extractable BARG.

Now, what Ali was saying is that the BARG that Zhengzhong showed you last time had the somewhere extractable property. It was extractable. Actually, let me tell you, we also will need BARGs that are somewhere extractable. I didn't write that here. Why didn't I write it here? And this is where the n--

AUDIENCE: [INAUDIBLE]

PROFESSOR: Exactly. The answer is any BARG, we can make it somewhere extractable by just adding a hash. So we can kind of-- we don't need to worry about somewhere extractability of a BARG, because what we can always do is add-- put a hash of all the witnesses. If you have a somewhere extractable hash, if you have a somewhere extractable hash, you can tell the verifier-- put a somewhere extractable hash of all your witnesses.

So take witness 1, witness 2, witness 3, up to witness k, hash them, and in the analysis, just jumping ahead, we're going to be extractable on all the locations corresponding to witness i. And now we're going to tell him-- we're going to say, give me a BARG. The BARG is not for the original language l. The BARG says-- the BARG that xi is in the language now says there is an opening for the hash value that [INAUDIBLE] the locations correspond to wi. If you opened, you would get a valid witness.

BARG that language. So instead of BARGing the original language l, BARG a little bit kind of upgraded language that, again, says there's a-- so the verifier will give you a hash of all the witnesses. And it will prove-- for every i, he won't prove just excising the language. He's going to prove for every i, I have openings corresponding to witness i. So the eight openings are valid. And BWI is in the language l. That's the BARG he's going to give you.

And now the fact that you're in the language means that it must be that if you take the trapdoor and look into it, it must be by soundness. It must be. That's what's sitting there, because you're binding. If you're going to be binding on that part, that's what's sitting there. It must be the correct witness. So that's kind of why, actually, we don't need to construct BARGs with extractability. We can always attach a somewhere extractable hash to them.

AUDIENCE: Yeah. I was actually wondering something about assumptions,. Do we have any constructions of BARGs from assumptions weaker than those needed to construct a somewhere extractable hash?

PROFESSOR: I see.

AUDIENCE: Which would then make it, I guess, interesting.

PROFESSOR: I see what you're saying. I see.

AUDIENCE: I don't [INAUDIBLE].

PROFESSOR: Yeah, I don't think so. I don't think so.

AUDIENCE: [INAUDIBLE] probably shouldn't.

PROFESSOR: Yeah. I don't know how to construct a BARG from an assumption for which I don't know how to construct a somewhere extractable hash. Yeah. OK. Any other questions?

So I want to get to the construction of the SNARG. But before I do so, I do want to spend a little more time on the somewhere extractable hash and say a little bit about the way we construct it, because I just want to make sure that you didn't miss a part from last lecture that I think was a bit said too succinctly.

So the high level idea of how you construct a somewhere extractable hash is using fully homomorphic encryption. So I don't want to get into-- I don't want to recall too much the definition, but I'll give you a high level. So the basic idea, which doesn't work-- let me first tell you the idea that doesn't work, because that's very natural. And I think maybe that's what you got from last lecture.

So the basic idea that doesn't work is to say that gen-- well, just the output, encryption, so we'll give you a public key and encryption of i with this public key. And the trapdoor, that's going to be the hash key. And the trapdoor is going to be the secret key. So just the hash key is going to be an encryption of the index.

Now, I want x1 up to xn. Let me assume for simplicity that N-- suppose N is a power of 2. Otherwise, you know, pad. So let's say big N is a power of 2. So it's some 2 to the small n.

Now, here's what I'm going to do. Let me-- so when I say encryption of i, let's think of it as bit by bit encryption. So it's encryption of i1 up to encryption of in. And now the idea is-- what we'll do is for each pair-- so I'm going to do kind of a Merkle hash. And for each pair I'm going to encrypt, so I'm going to do this Merkle hash, where for each node I'm going to associate a ciphertext.

And the output is going to be the ciphertext of the root. That's my hash value. The hash value is going to be a ciphertext of the root. And this ciphertext can be a ciphertext. It's an re-encryption of xi.

How do I associate? So of course, I can compute encryption of xi. This is a homomorphic encryption. Given x and encryption of i, I can do computation under the hood. So I can compute encryption of x sub i.

So because this is a fully homomorphic encryption, given x and encryption of i, I can compute encryption of x sub i. Of course, there's public keys. I'm omitting them for succinctness of notation.

This is because I can compute under the hood. The value of the route is going to be encryption of xi. But what kind of encryption? It's not any-- so let me tell you exactly how we compute this encryption.

So the idea would be-- and not only that, how do I open? I want to do a local opening. So the way I'm going to do it is as follows. For each pair, I'm going to encrypt only one of them. I'm going to either put here encryption of x1 or encryption of x2. Which one? Depending on i1.

So let's start with i1. If i1 is 0, I'm going to-- so I'm going to, let's say, choose x1. If i1 is 1, I'm going to choose x2. I'm going to choose the even-odd, so that the output will be the correct one. So I'm going to-- the i1, depending on encryption of i1, that's going to tell me which one to put here. 0, I put on the left, like the left children, or 1, I put the right children.

And then I do the exact same thing. Given these, whether I choose the left or right depends on the encryption of i2. If i2 is 0, I choose the left. If i2 is 1, I choose the right, and so on and so forth. That's the idea.

AUDIENCE: Is this like an OT?

PROFESSOR: It's like an OT. It's really like an OT. Exactly, exactly. I didn't define what OT is, but it's exactly like an OT. This does not work. It almost works, but it does not work. So let me tell you why it seems like it should work. And then I just want to mention the subtlety here.

So the reason-- so first, how do you generate the local opening? Well, you check-- so if you want to open xi, I'll tell you, give me x1 and x2 so I can check that this is correct. And then give me the hash in ciphertext here. And I can check that this is correct, because each ciphertext is some FHE valuation that depends on the two children and the hash key, which is encryption.

So in layer J, it depends on the two children and encryption of ij. Each time the node-- the ciphertext corresponding to some node is a deterministic function of the ciphertext corresponding to the children, and the hash key corresponding to that level. So encryption of, let's say, ij, because, remember, if I'm, let's say, in level 2, I have here a ciphertext. I have a 1 in ciphertext 2.

Then I do-- the way I compute the parent ciphertext is just a homomorphic evaluation that takes-- if i2 is 0, it takes ciphertext 2. If i2 is 1, it takes-- I'm sorry. So if it's a 1, otherwise, it takes ciphertext 2. So this is really just a deterministic homomorphic evaluation.

And so that's how I open. I just open all the relevant ciphertext and the verifier will check that the homomorphic evaluation was done correctly. So you can local opening-- you can do local opening, but it's not binding. Or I don't know how to prove that it's binding is the problem.

And probably, you can find a weird FHE for which it's not binding. I don't know-- maybe for natural constructions, it is binding. I don't know. I don't know how to prove it. But there's a problem with binding. And the problem with binding is that a cheating-- so it seems like it should be binding because, look, here it's sitting encryption of xi.

But the thing is the reason why it's not binding is because a malicious adversary, he may give ciphertexts that are garbage-- are completely garbage. They don't actually-- they're not legal ciphertexts. And if he gives ciphertexts that are garbage, I don't know why would there be binding. So I mean, maybe extract doesn't work.

Here's me. I'm a malicious person. i-- nothing-- no, you can't extract anything. I just want to ruin the binding. My whole goal is to-- kind of ruin binding here. I'm going to give you v that doesn't encrypt anything. It's a junk v. And I'm going to open it in two different ways. Maybe I can generate a v. Like I call it ciphertext, but it's just a bunch of bits that have no meaning-- cannot be decrypted at all. And maybe I can somehow generate valid openings.

And we don't know how to argue that you cannot do that. So in order to actually get soundness, I need to make sure that the things are well-formed, that these are actually valid ciphertexts, or what I open to is a valid ciphertext. And the way we get this is by making this a little more cumbersome.

So the way we get this is we actually don't use one public key. We use small and public keys. So for each layer, we'll use a different public key. And what we encrypt-- so what we do is we have public key j for j goes from 1 to n. So we have n public keys. And with public key j, we encrypt ij. But also, so when i has i1 up to in, we encrypt ij. But we also encrypt skj minus 1. I want to have the decryption. It's important to me to have the decryption key.

Now, so the important bit is the homomorphic evaluation I'm doing here, I need to know the secret key, because I'm not just going to say-- before I said, what's going on underneath under the hood? I'm saying, look, I have i1. I have two ciphertexts. Give me one of them. No, no, no, no. What I want to do is to say the following under the hood.

Decrypt this. Decrypt this. And give me only the relevant one. So that even if this does not decrypt-- so here's the point. Let's say I want to open to x2. He gives me x1. These are valid. So this is valid. Good. But now, ciphertext 2 can be completely not valid. Because when he opens, he gives me ciphertext 1 with these two. So I know this was valid, because this is a different computation.

And then I give him ciphertext 2. And I do this evaluation. And then he gives me-- so the sibling ciphertext that he gives me, they may not be valid. I want to ensure, even if they're not valid, even if they're not valid, this is going to be valid. The openings are going to be valid, kind of on the path, things are going to be valid, even though the siblings that are given in order to just kind do the-- verify the opening, even if they're not valid, the opening path will be kind of valid ciphertext.

So why are they going to be valid? Because the way that-- what is-- so now, what is the deterministic computation that's happening here? Or the evaluation, the evaluated ciphertext? The way we do the evaluation is the following. I decrypt this, and I decrypt this. And then I take the decrypted value. And I encrypt it with the next public key.

So the point is, even if this didn't decrypt-- maybe this didn't decrypt-- I don't care. And this is still going to be valid. So the point is, even if the sibling ciphertexts that are given only for verifying the opening, even if these sibling ciphertexts are malformed, if you give me the opening and the entire path, I get the guarantee that on the path, not on the siblings, on the path, things are well formed. And I mean well-formed ciphertext. These are ciphertext that decrypt correctly. And that's what I need for the soundness. So it's important to do the decryption. This is really crucial.

AUDIENCE: Are you doing it on each node? I didn't fully follow.

PROFESSOR: Again, what am I doing on each node? Good, good, good. So here's what I'm doing on each node. The evaluation function takes kind of ij secret key j minus 1 and two ciphertexts, ciphertext 0 and ciphertext 1. And what it does is the following-- decrypt ciphertext 0, so it decrypts with SKJ minus 1 ciphertext 0-- ciphertext b for every b. It gets some bit-- I don't know-- an alpha B.

And then it outputs one of them. I know alpha, b star, alpha ij. So it gets two bits, and he chooses one of them, which one depending on the ij you want. So what you-- the point is what you will get is really an encryption. So the point is, let's say if--

AUDIENCE: You're doing it homomorphically.

PROFESSOR: I do it under the hood. This is a computation that is done under the hood. Yes. So this is going to be encrypted in a box. But what I'm going to do under the box is to say-- so these, I have-- so sorry. These two are-- ij and skj minus 1 are in a box.

And the computation that I'm doing inside the box is I'm saying, I have these in a box. I have these in the clear. Inside the box, what I do is the following. I say, I have the secret key j minus 1 and these two ciphertexts. Let me decrypt both of them. I'm going to get two bits, alpha 0 and alpha 1.

Now, maybe one of them is bad, so bad. So maybe one of them doesn't decrypt. So the one that doesn't decrypt output bad. And then the output will be the relevant one that, according if it's 0, I'm going to output alpha 0. If it's 1, I'm going to output alpha 1.

And now what I can argue is on the path itself, everything has to be well-formed. So I'm actually going to get the correct ciphertext. And I can argue that by induction. And here, of course, this is well-formed, because this is just a deterministic computation that's done locally.

Now, the point is-- so let's say, like, I want to output-- I want to-- I'm binding on x1. x1 is what I want to open. So this must be well-formed. And now the point is, even if this is not well-formed, what will I do under the hood? I'm going to open this and open this. This is going to be the correct value. This is not going to be maybe the correct-- this may be bad. But this is the one I choose. And then I encrypt it. So I have it here.

And then, again, this may be bad. I'm going to decrypt it. It may be bad, but this is going to be good, and this is the one I choose. So one can argue kind of by induction that this is going to be encryption of x1. This is going to be encryption of blah blah blah. everyone encryption of x1 until the end. So there's no way you can open two different roads. You have to open x1.

AUDIENCE: And the verifier is also going to do the same computation to check?

PROFESSOR: Exactly! The verifier does the exact same computation to check. Yes. Yeah.

AUDIENCE: [INAUDIBLE] can only open the [INAUDIBLE] that you generated, right? This is not opening a general i.

PROFESSOR: Good. So this-- exactly. This shows that the specific-- so this-- this shows that the specific i that's encrypted here-- so there's one-- the hash key is binding on one i, on a specific i. The i on which you're binding, you can only open in one way statistically. And then by the index hiding, it means that you can-- for any j, you can only open one-way computationally, because you can't distinguish. Yeah.

AUDIENCE: Can you avoid this chain of keys if you are happy to just encrypt the secret key at the same [INAUDIBLE]?

PROFESSOR: Good, very good, good. Yes. You can avoid using many public keys. You can just use one public key. Encrypt the secret key, and rely on what's called circular security. That's another option, if you wish. Sorry it doesn't want to do it. But yes, exactly. Exactly. Yeah.

AUDIENCE: You can get collision resistance?

PROFESSOR: OK. So yeah, So you're asking, will you get a collision-- like collision resistance? And the answer is yes, because for this specific i that you-- for the specific i that kind of you encrypted in the hash key, you get a binding statistically. There's no way you can open two different ways.

AUDIENCE: What you have at [INAUDIBLE] is just the encryption of that index, right?

PROFESSOR: Right. So you're asking about a different j? Oh, sorry.

AUDIENCE: No, that you have two x's that have the same index. They have the same [INAUDIBLE], the index, but the rest are different. Will you get a different output of the hash if you take those two different x's?

PROFESSOR: Good, good, good. So if you take two different x's for which, let's say, they have the same x1. And let's say the hash key is binding on location 1, the value you would get if you hash them honestly will be actually very different. But because you use different x's, but so you'll get different values. But if you decrypt, these values correspond to a ciphertext. As values, they're going to look very different. But if you decrypt, you will get the same bit, which is x1. Yeah. Good.

So anyway, if you didn't follow the exact construction, it doesn't really matter. I just wanted to make sure that it was clear that you really need to do this decryption. Yeah.

AUDIENCE: The reason you don't need circular security as it is for this FHE is that each of these computations is simple enough that you don't need to bootstrap. Is that--

PROFESSOR: The reason? The reason I don't need a circular security is I'm never encrypting a secret key under its public key. So sorry.

AUDIENCE: I think he was asking why is it FHE that you're using, not [INAUDIBLE] unbounded.

PROFESSOR: Oh! Oh! OK. OK. You're saying-- OK. Sorry, sorry. You're asking me-- OK, got it. You're saying, while in general, if we want arbitrary FHE, we need to assume circular secure LW inside it, kind of. And you're asking, do we not neec to assume-- you're kind of going into the-- you're saying, do we actually need fully, fully, fully homomorphic encryption? Because if we do, then, actually, we need to rely on a circular security assumption inside, too.

The answer is we because the computation is very specific and very small depth, you don't need to-- there's no-- you don't need full, full FHE. You need FHE for very bounded depth computations. Yeah. Great. Thank you. Any other questions?

So let's SNARG. Well, let me just make sure I covered everything I wanted to.

AUDIENCE: After three months, we SNARG.

PROFESSOR: We SNARG. The last kind of two hours in class, last, we're going to SNARG. So now the SNARG is really simple. It's like as simple as one can-- like, essentially, you can do it. Done, we're done. So how do we-- we can go home.

Anybody want a cookie before we SNARG? I feel like these grandmothers that kind of constantly feed you. So here's the idea. The idea is very simple. Take any-- so SNARG.

So let me first say kind of the high level idea. The high level idea is the following. Here is what the prover will do. So the prover wants to convince that x is in the language. There's some verification circuit. So you have x.

Let's start with-- let's take x in the language. There is some-- for any x-- it doesn't matter-- for any x, you have a circuit, C sub x. And C sub x takes a witness and outputs 0 or 1, like it's a verification circuit. You can also think of it, if you want, C as taking x and a witness, and outputs 0 if it's not a valid witness and 1 if it's a valid witness. If it's a deterministic computation, there's no witness. You just take x, and output 0 or 1. It does the computation.

So when I talk about SNARG here, you can think of a SNARG of non-deterministic and a SNARG of a deterministic-- any language l. I don't even want to go too specific. But I have a language l, you know what does it mean a language? Well, there's some circuit. It takes the instance, maybe with a witness. And the circuit kind of does some checks to decide if it's valid or not valid.

Now, I want to convince you I have an x. I want to give you a SNARG that this C, an x outputs 1. Either that there is a witness that outputs 1, or maybe it's deterministic. So just C of x outputs 1. Here's what I'm going to do. What I'm going to do, I'm going to compute all the wires of this circuit. I'm Include all the wires, and I'm going to give you a hash of all the wires.

Now, there's a lot of wires. So it's going to be a shrinking hash. I'm going to actually use a somewhere extractable hash. So let's say we agreed on a hash key for a somewhere extractable hash. I, the prover, I'm going to compute all the wires of this verification circuit. And I'm going to give you the hash, the hash of the values of all the wires. The hash, what are you going to do with that?

So the other thing I'm going to give you is I'm going to give you a BARG, a proof. A BARG of what? The BARG I'm going to give you is that every gate here is satisfied. So I committed to all the wires. I can commit to anything. I mean, I had all the wires.

Now, I'm going to prove to you that for any gate, if you look at the two input wires to the gate and the output wire of the gate, they satisfy the gate. So for all-- every gate in the circuit, I'm going to prove to you that there exists an opening for this wire and an opening for this wire and for this wire, so that these openings are valid, and that the values that I open accept the-- kind of respect the gate.

And I'm going to prove to you that the output-- so I said that I hashed all the wires. I'm going to argue I'm going to open the output wire. So you see? And end output is 1. So again, what am I going to do? I want to convince you that the output here is 1. I'm going to hash all the wires-- the value of all the wires in the circuit. I'm going to prove that every gate is kind of satisfied. I have an opening that kind of respects each and every gate. And the output wire is 1. That's my SNARG. Yeah.

AUDIENCE: How is this different from GKR?

PROFESSOR: OK. How is this different from GKR? OK, so GKR is very different in many ways. First of all, it's interactive protocol. Now, you can apply Fiat-Shamir to it, but, essentially, in GKR, you take the circuit. You compute all the wires in the circuit. But actually, I don't BARG-- say, look, all the wires are satisfied. Instead, I'm going to-- kind of we're doing an interactive process, saying, I'm going to-- I argue the output wire is 1.

And then we do a little sum checker, some interactive process to reduce checking that value, to checking a value in one layer below. And then we do an interactive process to say, OK, if that was false, then the layer below needs to be false until we reach the leaves. And then you can apply Fiat-Shamir to that to make it a SNARG.

Here, we're doing something very differently. We're starting with a BARG. Now, it's true that the construction you saw for BARG uses Fiat-Shamir also, but some constructions don't use Fiat-Shamir, actually, for BARGs. There are various constructions of BARGs. You saw one of them, but there are others.

So we take any BARG. And there's no interaction here. Besides, maybe they're inside the BARG, but maybe the BARG doesn't have any Fiat-Shamir or anything to it. And we're saying just hash all the wires of the circuit and prove that everything is consistent. That's it. That's really what we're doing.

So let me write this down. So it will be-- so my ingredients that I have for my SNARG is-- so ingredients, maybe I'll write it here. So my ingredients is a BARG, and some are extractable hash. These are my ingredients. And I'm going to show you my SNARG.

So a SNARG has three algorithms. We have a gen algorithm. And here is what my gen algorithm is going to do. It's going to-- let's say the gen also has an input length. We don't need it. Here's what it's going to-- OK, let's say it has an input length. So this says, you're going to prove to me statements of length-- that the input x is of length n.

So here's what we're going to do. The first thing we do is we're going to generate a hash key for gen of the somewhere extractable hash. So generate a hash key corresponding to the somewhere extractable hash. This is going to be a security parameter.

n is going to be-- this is going to be the number of wires in the circuit C. This is the circuit that takes x of length n, possibly a witness, and outputs 0 or 1. So this is going to be the number of wires. And take any i. I don't know, whatever. i equals 1. Take any i in between 1 to n. It doesn't matter. Choose it arbitrarily, your favorite number.

So that's first thing. And then you also generate a CRS for the BARG. So you generate a CRS for a BARG. This is for the BARG. And this will be with input 1-- with the script 1 to the lambda. K, the number of statements is like the number of gates in the circuit. So K is going to be the number of gates in C, because we're going to-- remember, we're going to say-- we're going to prove that for every gate, it's satisfied. So usually, K and N are very similar. It's like essentially the same number. But let me just be clear that this is-- we're going to have the number of gates.

And the length n there-- let me call it n prime. We'll see what it is. We'll see what this n prime is in a minute. So we're going to BARG statements not of length n, but of a different length, n prime. We're going to see what that is. But the output is just a-- It was going to be called the CRS BARG. So you output CRS, which is equal to hash key and CRS BARG.

So that's how you generate. So the key generation, the CRS has a CRS for the BARG and a hash key. Now, we're going to-- what does the prover do? So the prover gets as input x. Maybe a witness, if it's an NP. Maybe there is no witness. Let me say, generally, with a witness. But actually, the BARB we're going to construct today is going to be for deterministic languages. But let me write it also, because I'm going to show you a general framework.

So P is going to get a CRS, an x, and maybe a witness if it's an NP. And it needs to generate a SNARG. So what does it do? Compute all the wires in the circuit. So x is fixed. Let's think of x as, like, that's not part of the wires because it's fixed.

But let's compute all the w1 up to wn, which is the wires of the circuit. You can think of the first n as the actual witness. Let's say if you have a witness and it's of length m, w1 wm is the witness, but then the rest are the wires. So I'm thinking the wires include the witness and everything above. And if there is no witness, then it doesn't include a witness.

So I compute. These are all the wires in C, except x, except the input wires corresponding-- the input wires corresponding to x, because that's-- kind of we know what it is. The verifier knows it. So you compute all the wires.

And then you compute the hash of-- you compute z, which is a hash, the eval with hash key of w1 up to wn. So you hash all these wires together. So the verifier computed all the wires of the circuit, hashed all the wires.

Actually, let me just say-- sorry. When you generate a key, I said use a somewhere extractable hash. Yes, exactly. But you don't need to just use one index. So let's use it with a few index, il. Now you can ask, what is l? We'll see. You can think of it as-- a few l's, you can think of polylog l. The bigger the l is, the less succinct the SNARG is going to be.

So the SNARG is going to grow with l because the hash key grows with L. It's like L hash keys, essentially, because we're repeating. So here I'm going to be binding on l locations. The hash value grows with l, because you're computing, essentially, l hash values. So the bigger l is, in some sense, the more secure we're going to get. We'll see. But the more non-succinct it's going to be. So l now is going to be a parameter. And when we talk about the analysis, we'll see which l we want. Yeah.

AUDIENCE: Is there any hope for getting a sublinear dependence on l, like, any other construction instead of the direct product construction?

PROFESSOR: So in V, there's no way to get sublinear in l, because you need to extract. It's like there-- it must have information. So in v, there's no way to have. In the hash key, on the other hand-- OK. If the i, i1 up to il, can be shrunk somehow, if there is a succinct description of i1 to il-- for example, in the case of a witness, I want to be extractable on witness i. These are consecutive locations, so I don't need to write location i, i plus 1, i plus 2. You can just say i, and go from i10. There's a very succinct description.

In the case where there's a description, you can hope to get the hash key smaller. And we do know how to get the hash key smaller. But in general, for the v value, there's no way-- information, theoretically. Great question.

So you compute the hash value of all the wires, and then you add a BARG. So the last thing you do is you compute the P BARG. You take the BARG prover. And here's our BARG prover. You give it CRS of BARG. And you have x's and witnesses. Now, what are the x's and the witnesses?

I want to use-- I want to argue I'm going to generate a bunch of x's and witnesses and generate a proof. So what is it that I'm-- so I'm going to give you x1 star xk star and w1 star. I'll tell you what these are in a second-- wk star. So this is going to be pi. This is going to be compute pi BARG.

So you BARG k statements. But what are these statements? So let me just tell you what these and these are. So x star j is-- essentially, it's going to be hash key value nj. That's x. So what I'm going to prove, that for each gate j, I have an opening to the input wires of the gate and the upper wire of the gate with-- I have an opening with respect to v that satisfies the gate.

So the xj is going to be-- well, I have the hash key. I have the hash value and gate j. And the witness for gate j is going to be opening. So I'm going to have a bit, like a left child, right child, output child-- not child-- output, the parent. So left child, right child, and parent of the gate opening, rho of left child, rho of right child, rho of the output. That's going to be a witness. And it's a valid witness. If B0 rho 0 are valid opening of the left child of kj.

b1 rho 1 are a valid opening of the right child of kj. b out and rho out are valid openings of the output wire of kj. And these satisfy the gate. So if it's an AND gate, this and this is this. That's what-- that's going to-- so a valid witness is a six tuple like this that all these openings are accepted, and these bits satisfy the gate.

And our valid opening with respect to which wires, the wires that I've defined by the gate. So think of it, we have this circuit C. The gates are kind of numbered. And for any gates j, we know there's-- we know the number of the input and output wire, because this is uniform. Everything of the circuit is uniform. So for every gate J, it corresponds to-- I know i, wire i, wire j, and wire k. And so these openings should be with respect to wire i, wire j, and wire k.

AUDIENCE: Do you use the same hash key for all of them?

PROFESSOR: Same hash key for all of them. Yeah, because I used one hash key. I mean, one hash key that's binding on l locations. But I'm now I'm thinking just one hash key that's binding. How I construct it, we'll abstract that for now.

But yeah, I have one hash key that's binding on l locations. I hash all the values of the values of all the wires with respect to this specific hash key. And then I add a BARG for any gate j that I have an opening for the input, the two input children, the output wire of gate j that satisfy the gate. Yeah.

AUDIENCE: So the only difference between the xj is the index j itself. So it's like, what are the index [INAUDIBLE] things?

PROFESSOR: Yeah, yeah, yeah. Good, good, good. So you're saying, where's j here? Where's j? You're saying--

AUDIENCE: No. More like x, j star, and xj--

PROFESSOR: The only difference is j. So this is an index kind of language. This note, this is kind of-- it's the x1 and x1 star, x2 star, are the same hash key. vk1, 2, 3, 4 up to k. So it's kind of an index language. We're going to get back to it when-- because we want the efficiency of v. So we'll talk about that. But yes. Any question about the prover here?

So now the verifier, what the verifier does is-- so now let's write the verifier somewhere. Oh, that's sad. I wanted to keep this. If you don't mind, we'll write the verifier here. So what does the verifier do? V takes as input a CRS, which is a hash key and CRS BARG. That's the CRS. And it takes an instant X and a proof.

What is the proof? A proof is a value and a BARG, a hash value and a BARG. So the verifier, he gets the CRS, which is of this form. He gets an input x. And he gets a SNARG. A SNARG is-- oh, did I say what it outputs? Sorry. Sorry. Compute, output. v in pi BARG.

So now he takes the SNARG, which is a hash and a BARG. And it just outputs 1, an output. The output of v BARG, he accepts if and only if the BARG is accepting. What do we mean the BARG-- v BARG takes CRS BARG and x1 star up to xk star, which are essentially an index language corresponding to hk, v, and j. So it accepts if and only if the BARG is accepted. That's the SNARG.

One thing I-- so again, you generate a-- again, what is the SNARG? In the CRS generation, you generate a hash key for the somewhere extractable hash, and you generate a CRS for the BARG. The prover computes the values of all the wires in the circuits, hashes it using the hash key to generate a value, a hash value V, and then he proves-- it gives a BARG that all the gates are satisfied. Namely, for every gate he has an opening valid that's satisfied-- that respects the gate.

And the output gate is 1. That's another thing. He also proves that-- the BARG also proves that the output gate is 1. So for the output gate, you prove it respects, namely-- you prove it's 1. Good. So that's-- and how do you verify? You just verify the BARG.

Now, here's one important thing to note. This v BARG takes as input the BARG of length k. k is like the number of wires in the circuit. We do not want the verifier to run in time k. That would be the size of the circuit. No, we want the verifier to be very efficient.

But that's OK, because he actually doesn't need to run in time k, because, actually, it's what's called an index language. Namely, x1 up to xk star, essentially, it's like an index on hash k and v. So you can generate them. You can express them very succinctly. You can express them by hash key V, and all the index between 1 and k. That's it.

So because they have a succinct representation, the runtime of the verifier only grows with hash key nv, polynomial with hash key nv. He does not need to grow with k. That's kind of the efficiency property of index languages. And that's very important. Without it, we can't use it for the SNARG.

So in terms of efficiency, if the BARG has good efficiency, let's say the dependence is polylog on k, then the v is going to be-- so what is the-- the SNARG grows with v and with a BARG proof. The BARG proof is very-- it grows polylogarithmically with k. It's like poly log the circuit size. It's very efficient.

v grows with the number of indices we're binding on. So it grows with l. So it grows with l. We'll try to minimize l. We ideally want l to be polylog k. So that's what we want. We'll see what we get. But ideally, this should be polylog k.

And the runtime of the verifier, because it's an index language, is also going to be polylog k, assuming a-- sorry-- and he needs to-- he does run in time x, because he also needs to check this pi BARG is pi BARG comma of x, because when you check the validity that each gate is satisfied, the gates in layer 1 depend on x. So the gates in layer 1, the input wires are x, xi, xj. So the BARG needs to know x. The verifier, of course, needs to know x. But that's it.

So he doesn't need to run in time-- linear in x. But the BARG itself is very short. So this log is very short. The verifier, of course, needs to run time x. It needs to read the input, but that's it. So any questions?

So let's take a break, a cookie break, a fun break. And let's start, again, like 10 minutes, and then we'll talk about the analysis and when-- but this is all-- but that's done. We're done with the construction. This is your SNARG. We built up to it. Here it is. And then we'll analyze it after the break.