[SQUEAKING]

[RUSTLING]

[CLICKING]

ZHENGZHONG JIN: So here, so from this claim we can show this is in this relation r. So in fact, we also need a CI hash for this relation r. So in this contraction, we need a CI hash contraction for this relation r. So this follows from the-- so for this kind of CI hash, it follows from the work.

So it almost can handle all of the sparse relation up to some parallel repetition. So I will not go into the detail of this work. So if you are interested, you can read it yourself. So this is the work-- so it's a CI hash for efficiently verifiable product relation. It's a beautiful work by-- so this is the [INAUDIBLE]. OK. OK. So finally-- so let me ask a question. OK. Yeah?

AUDIENCE: Could you say again, why we need-- like, why the CIH we saw in last week's class isn't good enough?

ZHENGZHONG JIN: Oh, right. Good question. So last week, the CI hash you saw is for searchable relation. So that means the relation-- so the relation here. So the CI hash relation here. Right. So the relation here need to satisfy the searchable property. So it means-- so this beta-- so for any fixed alpha, there exists a unique beta such that this alpha beta is in this relation r.

And moreover, there is an efficient algorithm, some efficient algorithm f such that it can take this alpha and the output, such as beta. So that's a efficiently searchable mean. So here, it's possible that for a fixed alpha there might be multiple beta that can satisfy this relation. Because for a fixed PCP, there might be multiple queries that can make the PCP be accepted. Yes. Good question. Yeah.

So what-- HR shows that if the sparse relation has an efficient wildfire algorithm then most of the-- then you can build a CI hash for them up to some parallel repetition. Roughly speaking, it's a-- yeah. So now, we have a contraction of SNARGs, but let's see what's the proof size for this SNARG. So the proof size consists of the first message and the third round message.

So the first round of message consists of l, l for hash values. So you can think of each hash value is also [INAUDIBLE] ciphertext, so it's a poly lambda bits. So in total the first round of message consists of poly c times poly l, but it is independent of k. And the third round message is k times poly log c times poly l because, for each in the Celeron message, for each instance you send a q PCP answers. So the size of q is lambda times poly log c.

So in total, you send a k of them. So in total, it's k times this. So this is already non-trivial because it's a-- so this number, it could be much smaller than k times k times w. Because the second term is almost order of k, and the first term is almost c. So this is already non-trivial, but it's still large.

So the big issue is, if you look at the verification time, so the verification time is you will see is k times poly c. The reason is the verifier need to verify the PCP answers for each of the instances for x1 to xk, so in this step. So each PCP verification need to take c time, and in total, it need to take k times poly c. Yeah?

AUDIENCE: Why is there a poly lambda at the end?

ZHENGZHONG JIN: So that's because each query is a-- so each query, so if you remember-- yeah, so here. So each query size is a lambda times poly log c. So it's-- yeah.

AUDIENCE: So they're not just querying single bits. They're querying like blocks of links.

ZHENGZHONG JIN: So here, we have a lambda term because we need to do the parallel repetition. So you take PCP with constant suddenness arrow, and then you amplify it lambda times by parallel repetition.

AUDIENCE: And we want to amplify it because we want complete, like perfect completeness or something?

ZHENGZHONG JIN: Yeah, so you can have PCP with perfect completeness. So you don't need to worry about when you do the parallel repetition. So verification time is still very large. So in fact you don't have any savings if you do this directly, so on the verification time.

So let's consider for a general SNARGs for batch NP, or BARGs, so what's the verification time? So remember, in a bark, the verifier need to read at least the CRS and all of the statements to verify the proof. So the verifier need to at least read all of the instance. So it seems that there is a lower bound on the verification time. The lower bound is just at least K times X.

So next I will show you how to reduce the verification time and the proof size of BARGs. So we will consider a variant of BARG, and it don't have this lower bound on the verification time. So I will define a new notion of BARGs. Let me erase this.

So this notion of BARGs is called the index BARG, or you can also call it SNARGs. for a batch index. So in the original definition of BARGs, the verifier need to read all of the instance, because the instance are non-uniform. So in this notion, we will consider a special case when this instance are uniform.

So intuitively, it means there exists some circuit U such that the I-th instance is generated by using the circuit U and the index I. So intuitively, it says the instance can be described in a uniform way by only using the index. And if this is the case, then what the prover will try to prove is X1 to XK is all in this language L.

So now, it is equivalent to say, so for each index I, there exists a witness WI such that this circuit C and the UI WI equals 1. So then we can define. So essentially, this you can view this as a new NP relation. So the new NP relation is defined as follows.

So the instance is just an index, and it verifies there exists a WI such that. So this is just a rewriting of this, right. So, right. And you can view this as a new NP verification circuit that some C prime so it only takes WI, and I and WI as input. And it internally compute this U and verify C.

So this inspires us to define a new notion called index BARG. So, an index BARG definition-- an index BARG is a BARG for this kind of relation. So it is for this kind of language where the language is just some general NP language, but the instance is treated as an index for this language,

So the prover, we still have a CRS. And the prover tries to convince the verifier that all of the indices from 1, 2, et cetera, to K is in this language. So the honest prover only need to read index K, and it has a witness from W1 to WK for each of the index. And the verifier only need to take K as input.

So now the verifier, so this K can be represented as log K bits, because it's just an index. So what's the best verification time we can hope for? Anyone know?

So all good. So the verifier only need to read this index K and the proof. So the verifier can be as small as verification time, only need to be some log K, can be as small as this much, so only depends on log K and the lambda.

And for this index BARG, we also require the completeness and the soundness in the same way as before. Any questions? Yeah.

AUDIENCE: How should I think of this U? Or should I think of it as being part of the instance or something globally defined?

ZHENGZHONG JIN: So you can always absorb it in the relation circuit. So you can think it's part of the relation.

AUDIENCE: Right. I guess, are we going to pay for the size of the description of U anywhere? Or is that a?

ZHENGZHONG JIN: Right, right, good. So the verification circuit, in fact, also depends on C, because the verifier needs to read the circuit.

AUDIENCE: Yes. And is there like a unique instance in this problem, then, like, where you're saying K itself, is that OK?

ZHENGZHONG JIN: So now, the point is now the verification time only depends on linear in K. So previously, you need to read all of the instance. And now you, the verification time only grow log in K.

AUDIENCE: Are we supposed to think of C as part of the input?

ZHENGZHONG JIN: C as part of the input?

AUDIENCE: So like, or I guess C prime, which stores U in it.

ZHENGZHONG JIN: Right, yes, yes, right.

AUDIENCE: So, that is part of the input. So V still has to read what C prime is? So doesn't that take a lot of value?

ZHENGZHONG JIN: So it depends on how you define it. So you can define it for a fixed C, and then you have such a proof system. Or you can define a proof system for any C, then you need the verification time to grow with the circuits, size C.

AUDIENCE: Is it defining it for a fixed C not that helpful, because you're just solving batch SNARG for a fixed set of X's.

ZHENGZHONG JIN: But you can choose C to be universal circuit, then it's a-- yeah. Things in general. Good question. So in fact, I think, if you take universally, you need to-- if you use the U, maybe. It's good to think of C as the input.

So in the remaining time of this lecture, we'll prove the following theorem. So there exists an Index BARG from LWE. And the proof size is poly log, so it will grow log in K, C, and lambda. And the verification time is log KC lambda. So both of them almost match the best you can hope for, up to some polynomial.

So I want to emphasize here that, in fact, you can, if you know how to build this index BARG, then you can build BARGs with almost the same verification time and proof size. So in other words, in fact, a batch of instance are without loss of generality uniform, in some sense.

So, what I will show you is, in fact, if you know how to build index BARG, then you can use it in a generic way to build BARGs. So the reduction works as follows. So now we want to build a BARG for a bunch of instance from X1 to XK. So the idea is we will first build a Merkle hash of all of the instances. And we have some hash root.

So now we have the prover send this hash root to the verifier. And then we have the prover to show that for any index I, there exist a local opening, row I. So the local opening, remember, is the root to leaf path for each of the leaf.

So there exists a local opening, row I, for XI, such that the local verification will pass for this XI local verification. So the local verification algorithm will take the hash root and the index I and the XI, which is the open the value, and also this local opening, which is the root-to-leaf path. And moreover, there also exists a WI such that the original circuit is satisfied.

So the idea is you just hash all of the instance from X1 to XK, and you send the hash value to the verifier. Then you prove for each index I there exists an opening XI to this hash. And this C of XI is satisfied.

So in this way, so you convert a batch of NP instance to a batch index instance. So now, the statement only grows. So it proves something for each index I, so you can view this part. So you can view this part as some circuit C prime. So C prime will take index I and the witness row I, WI, and check the two things.

Then you can generate an index BARG for this instance for each of the instance from X1, from 1 to K. And then you generate an index BARG for this kind of instance. So now you send this to the verifier, and this is your BARG. So if pi is small and H is small, then your total proof size is also small. So that's the idea.

So I will come back to this point later. Because the proof of soundness of this transformation is not trivial, because, in fact, you will see this is not-- so the non-adaptive soundness previously is not sufficient to prove the soundness, because this circuit will depends on this hash, and it's chosen adaptively by the adversary. So we will come back to this point later to show how to prove the soundness of this transformation.

So next, I will show you how to build this index BARG. So our starting point is still this protocol. So, as I just said earlier, so the first round message is small because it only depends on C, but the third round message is large. So how can we further compress the third round message?

So in the context of index BARG, all of the instance are just indices. So I can replace them with 1 to K. And the idea is we're going to delegate. We're going to remove this round of message. And so we're going to delegate the verification of this message to the prover.

So the prover will have this open the query, the columns in his hand, but he is not going to send them to the verifier. Instead, the prover is going do the verification by itself and prove to the verifier those verification is indeed correct. So the conjunction work like this.

So the prover still sends this H1 to HL to the verifier. And the prover is going to compute this PCP query using the CI hash. And then the prover and the verifier is going to run another protocol, so another SNARG.

And this SNARG on the prover side, he's going to use the columns, open the columns as the input to this protocol. And in the end, the verifier is convinced the third round message verifies.

So what does this mean is the prover is going to prove to this verifier in this protocol the following language. So there exist log columns Q such that two things. So one is for each column, for each Q in the query, the set, the hash is computed correctly, is consistent with the first message. So HQ is the SSV hash of the column. The Q is column. So I omit the SSV key here.

And the second, for each PCP, for each instance, so for any I in K, so here the instance is just the index I. The I-th instance XI is just the index I. So he will verify this. So the PCP verification of I Q under the Q's column, the I-th coordinate of the Q column. And we collect them together.

So this is a statement this protocol wants to prove. And the prover is going to supply it with the witness, which is this. So that's the high-level idea.

But to achieve this, so suppose we can rewrite this statement as a batch index statement. Then I can apply the same construction again on top of this statement. So we can apply the same thing again to this new statement. And I can do the recursing. So that's the idea.

But the issue is how do I represent this as a batch index language. So how do I represent this as a batch of index language? So how do I represent it as a batch of index?

So now you can see, maybe you think, OK, so this is almost index because index language. Because at least this part-- this is clear for any index I. This is for any index I in K, there is something. So this is similar to our index language, but this part is not clear.

So the reason is Q is very small, but the NP witness is very large. So if you treat this Q as an instance, then the witness size is K, because each column has K entries. So each column contains the PCP coordinate for each of the instance, so the column size is K.

But if I treat this column, column Q, as a witness, then when I do the recursion, the proof size will grow with the size of K because, again, I will have this kind of hash value, some L prime. But those are H1 to L prime, H1 prime to L prime prime. So they are for the new protocol and the new instance. So they will grow with the size of K. So that's too large. We cannot afford that.

So next, I will rewrite this instance as a batch of statements. And the statement size will be K, but the NP witness will be small. So how do I rewrite it?

So let's look at what this do, this verification do. So for each of this verification, we have a column Q, and its size is K. So it will recompute this SSV hash and check if it is the same with some HQ, which is the hash that was sent in the first round.

So the idea is that suppose this SSV hash, suppose we have some more structure of this SSV hash, then I can local verify it. So that means, for example, if the SSV hash is built as a Merkle tree, then I can replace this verification by verify that for each index I in K, there exists a local opening, rho IQ, so rho IQ is the local opening.

You can think for each index I, I have an entry. The entry is column Q, the I-th position of column Q, and I have a local opening for that, rho IQ. So I only need to show there exists such a local opening for this hash. So here, HQ is the hash value. So for HQ, I only need to verify this. So now I get a batch of NP instance.

So let me rewrite this. So suppose I have a SSV built in the Merkle tree way, so the protocol will work like this. So initially, we have H1 to HL, and we have this Q.

So in the new protocol the prover is going to supply it with query the column for this Q, and also a bunch of local openings for each coordinates in this Q. So this protocol is a new index BARG protocol. It will prove for a new instance L for a new language L prime. Let's denote it.

So L prime, it contains all the index I. And for each index I, I need to show there exist a witness which is column QI. So this is the I-th PCP answer. And they are corresponding. I also need this rho IQ.

So first I need to verify the PCP answers. And second, I need to verify the local opening is correct. So there is some local opening verification algorithm. The local verification algorithm will take the hash root for each Q. And so I need to verify this for every Q. And this local verification algorithm of SSV will verify row IQ.

Any questions? So this is the new instance, new L prime. So for each I-- so the prover need to prove 1, 2, et cetera, to K are all in this L prime. Any questions?

AUDIENCE: I just think we're going to recurse with this? And the idea is that the number K has remained the same and the witnesses are not throwing anything.

ZHENGZHONG JIN: Good question. So if you do this recursion, it will loop forever. Because from the beginning, we have K instances. And then in the next level of the recursion, we still have K instances.

AUDIENCE: [INAUDIBLE] bunch it up, maybe, like, you do it [INAUDIBLE].

ZHENGZHONG JIN: Exactly. So that's what we will do next. Any questions?

AUDIENCE: So HQ is a hash of statements?

ZHENGZHONG JIN: Sorry?

AUDIENCE: HQ.

ZHENGZHONG JIN: Right. So HQ is part of the statement.

AUDIENCE: Is it the hash of the statement X1 to XK?

ZHENGZHONG JIN: You mean, where is HQ comes from?

AUDIENCE: Yes.

ZHENGZHONG JIN: Good. So HQ is the same as HQ here. So initially, we have a bunch of statements from 1 to K, and each of them has NP witness from omega 1 to omega K. And we compute this PCP matrix. And for each column, we hash it to H1 to HL. And HQ is just the Q's column of this H1 to HQ, the index Q among these H1 to HQ.

AUDIENCE: Can I see the local opening version of the SSB is some variant of the FHE fitting with a Merkle tree size?

ZHENGZHONG JIN: Good, good. So that's what I'm going to talk about next. So now you may ask, so how do we construct such a SSV with a Merkle tree-style construction, because the construction I just showed you doesn't satisfy this property?

So this is SSV. So SSV with this local opening property, so the idea is, if we build a hash like a Merkle tree then we have this local opening, so for example.

The issue is how do we still have somewhere-- this somewhere so that is a binding property. So at each node of the tree, we are going to peek under the hood of FHE one of its children. So you can think, initially, we have some X1 to XK in the clear. So at each level, we're going to pick either its left child or right child for each node in this level.

For example, in this level, so suppose this is your index I, then what we can do is for each node in this level we're going to pick the left child. And at this level, we're going to pick the right child. So at each level, we're going to have some homomorphic encryption of some bit, so some bit, B1 and so let's say this is B2. And each level, we're going to homomorphically evaluate this using this B1 and B2 to pick the children under the hood of FHE in the same way as before. So this can give you SSV with local opening.

So one property we further require about this SSV is the local opening correctness. So we require this property because, remember, in the soundness proof we need to ensure if the proof is accepted in the third round, then the proof contains some opening of the SSV. And we need to show this opening must be consistent with the SSV extracted value.

So originally, this is easy to ensure because the verifier sees the entire opening. But now it's not clear whether this can still hold. So now the verifier only sees a local opening of the SSV. So it does not see all of the X1 to XK. It only sees the root-to-leaf path of some XI. So we need a further local opening correctness.

So the correctness of local opening says this SSV with local opening has some local verification algorithm. So it will take the hash root of H and some index I, and XI, and the root-to-leaf path for this XI. And it can decide whether to accept or reject.

And the local opening property says that for any index I, H, XI, and rho I, so if this local verification passes, then it must be case that the extracted value of the hash equals this XI. So here, so the local verification also need to use the SSV key to check this.

So here, if this is true and the SSV key is generated for the index I, if the local replication passes, then the extracted value must be XI. So this is not trivially correct. This is not trivially hold if you use just this construction because a malicious file, it will contain the root-to-leaf path to this XI.

But it also needs to contain the siblings of this. So in this authenticated path, it also need to contain the siblings of each node in the path. But those siblings may not be well-formed FHE ciphertext. So indeed, you need to further add some bootstrap mechanism here.

So at each level of the FHE, you need to use a different FHE key. And at the next level, you need to homomorphically decrypt the FHE ciphertext at the previous level to ensure it's a well-formed ciphertext. So I'm not going to the details, but you can fix this easily by your own. So now, any question with this construction?

AUDIENCE: Why are we now using SSV in a Merkle tree style instead of?

ZHENGZHONG JIN: Good question. So if you directly use SSV in a Merkle tree style, maybe there are some ciphertext sites blow up. So each time the ciphertext size is lambda times larger than the.

AUDIENCE: I mean, in this new version index BARG, why now when we hash the column? We hash it in the Merkle tree side instead of hashing it in one?

ZHENGZHONG JIN: You mean, we don't hash it entirely. We hash in the Merkle tree style.

AUDIENCE: Yes, yes.

ZHENGZHONG JIN: Yes, yes. Yes, that's what we are doing. Any questions? So if you do this construction directly, then this construction will loop forever because the recursion will never end. So initially, we have K instances. And at the next level, it still has K instances.

So next, what we will do is we will use a 2-to-1 trick to reduce the number of instances each time. So let's denote this circuit. So this part of the circuit has some C prime. So C prime will take some index I and the witness WI prime. So the witness WI prime corresponds to this part. So this part is the WI prime

So the 2-to-1 trick is this. Suppose you have K instances. So now you want to verify all of them So you have instance from 1 to K, and their witness are W1 prime to WK prime. So the prover wants to prove to the verifier that all of them are correct, are in this language L prime.

So what do you do is you look at any two adjacent indices instances and just combine them. So now, the combined instance is still 1, but the witnesses are concatenated together. So you have this W1 prime and W2 prime. And the next two instances are 3 and 4. So you have these 3 prime and W4 prime. And the final one is the K and WK.

So you combine each adjacent one to get the second instance. And you do this for any two of them. So in the end, you have K-by-2 instances. But the cost of doing this is the size of the witness will blow up by a factor of 2.

So let's say the original language is L prime, so it contains of I such that there exists WI prime. So C prime is the verification circuit of L prime. Then here the new instances is the following. So it defines a new language, L double prime. So double prime is still contains of I, but the witness are 2I minus 1 and 2I and W2I, 2I prime.

So this concatenation witness is the new NP witness for this L double prime. And then you need to verify C prime of this 2I minus 1, W2I minus 1 is 1. And also, 2I and the 2I is 1. So by doing this, the size of the number of the instances will halve each time when you do the recursion.

So in fact, we will prove this for L double prime. And we only prove for 1 to K by 2. They are all in L double prime in the recursion. So that's what we will do.

So now the question is, at each level of the recursion, the number of instances will halve. So at the end, there will only be one instance. So what will we do at that point? Anyone know?

AUDIENCE: We wrap it and we use that?

ZHENGZHONG JIN: ? Yes, yes. So we directly send the witness, and the verifier can verify it.

So this is the construction, and let's calculate the proof size for this construction. So initially, we send this H1 to HL. So the size of that is poly C and lambda. And we have this Q computed by the CI hash.

So at the next level of the recursion, we will also have this first round message for that. So let's denote it by HI prime. So what's the size of this message? So remember, this is the index BARG for L double prime. And it has the verification circuit, C double prime. So what is the size of this message? So this is efficiency.

So in the next level of recursion, we will do the same using the same protocol. So originally, the NP verification circuit is C, so we have first round message this much. So in the next level, the NP verification circuit is C double prime. So the size of this message is poly C double prime lambda.

But there is an issue if you do this recursion directly. So let's examine what is the size of C double prime. So each C double prime will contain two copy of C, C prime, so C double prime. So this is the C double prime. So it will take index I and this as the witness, and verify these two things.

So the size of C double prime is roughly 2 times C. So what is the size of C prime? So C prime is here. So the first part of C prime is the PCP verification. So here we will have an issue because the PCP verifier need to know what is the NP relation. So the PCP verifier runs in C time. So we will have an issue here because the C prime is roughly C, at least C.

So if we do this directly, so you can think, at the top level, we have this poly C. And at the next level we have this poly C prime, which is 2C. So this is 2C. And at next level of recursion, we'll have 4C.

So we'll have log K level of recursion, because each time the number of instance halves. So the depth is log K. So in the end, we have K times C, which is large. So the proof will grow with K times C, which is large.

So the final idea is we are going to use PCP with some special property. So this kind of PCP has this online/offline verification property. So this is called PCP with faster online verification. So this PCP has the property that, so for any PCP prover and verifier, so given any NP instance X and W, you can generate a long proof. So this is just a normal PCP, but the verification algorithm of the PCP has more structure.

So the verification algorithm, so originally the PCP verifier need to verify for X, Q, and the PCP Q. So now the PCP verifier can be decomposed to two parts. So first, so there is an offline preprocessing phase. So the preprocessing phase only depends on this relation circuit C. So this is PCP for the language L.

So the preprocessing phase will take the relation circuit as input. And it will use the query to generate some state. And this state will only grow-- the size of the state is polynomial in X. So the point is, so in some case when the relation circuit is very large, so the preprocessing phase will run in the size of the relation circuit, but it will output a very short state.

And the secondly, in the online phase, there is an online verification part. So the online verification will only use such a short state and X and Q and the PCP answer to decide whether to accept or reject. So the point is, it's a polynomial time algorithm, which means it only depends on X and the size of Q.

So you can imagine if C is a very big circuit and the X instance is small, then this means the preprocessing phase will take most of the time, but the online verification will be very fast. It only depends on the instance size and the size of the query. So this property is satisfied by most of the algebraic construction of PCP.

So any questions? So I will not go into the detail. So in fact, this property is used in many practical SNARG construction. Because in the practical SNARG construction, they also need to use PCP or some form of the PCP, such as linear PCP.

So in this construction, you also have the problem that the relation circuit is very large, but you want the verification to be small. So what they do is they can do some preprocessing on the relation to generate some short verification state. And given the verification state, you can verify the statement in a short time.

So now we will use this PCP further with this property. Then we can further modify the construction to only do the online verification here in C prime. So what we will do here is we first generate the CI hash, using the CI hash to generate Q. And then we do the preprocessing to this Q to generate some short state. And this will use the relation circuit C.

So we can do this because we use the same PCP query for all of the PCPs. So for all of the PCPs, they are using the same PCP query, so I can do this preprocessing. So once I have this a short state, I can hardcode it in this construction of C prime.

So I will replace this with the online verification. Sorry. So this is a local verification. So this is online verification of the PCP. So it will take the state of the rest.

So now this will solve this recursion issue, because now the circuit C prime will not depend on C. Instead, it will be polylog in C. So you can check here because C prime, it will only involve online verification. So it's a polynomial in the length of I, which is log K.

And the local verification of the Merkle hash is also small. So it's also some polylog in K. So you can check. So this is a polylog of K and lambda. And you can take this recursion inside of this analysis and see, finally, the total proof size is poly log in C and poly log in K and polynomial in lambda.

AUDIENCE: Plus, the length of one witness or-- oh, I guess it's also poly in C. Like the base case, you said the witness, right?

ZHENGZHONG JIN: Yes, yes.

AUDIENCE: That's already populated.

ZHENGZHONG JIN: So in fact, after you do this once, at one level of the recursion the witness is already very small, because the witness is just the PCP opening. So it's polylog C. So finally, at the end the final level witness is also very small.

So I probably don't have time to cover all of the security proof, but I will show you the most important part of it. So how do we prove the security of this construction? So now the construction is here.

So suppose we have some security of this index BARG, then maybe we can use it recursively to prove the security of the entire construction. But we have an issue here. So the issue is L double prime, it will depend on the first message. So L double prime will use this L prime, so this C prime.

So this C prime will use the hash root, this HQ. So this hash root comes from this hash value and of the H1 to HL. So this means the adversary can potentially choose the next statement, depends on the CRS because CRS is first send to the adversary. So this is not captured by the non-adaptive soundness before. So we need a new definition of soundness.

And the issue is that we don't know how to achieve fully adaptive soundness. So the adaptive soundness means that the cheating adversary can choose the instance, depending on the CRS. So what we prove is some semi-adaptive soundness.

So this soundness notion is not only important in this security proof, but will also be crucial for your next lecture. So when you compile this batch index of SNARG for P. So it's defined as follows. So it says that it allows the cheating prover to choose the relation circuit, but it cannot choose which index to cheat.

So let me define it. So we say, index BARG is sound if there exists if. So first, I need another key generation algorithm, CRS generator. So I'll call it generate bar. So it will take the security parameter as input and index I star, and will generate some CRS star. And this CRS generator is indistinguishable with the real CRS.

And furthermore, for each non-uniform PPT adversary cheating prover, P star, we need to have the following property. So when P star receives the CRS generated by this generator star, it can choose as relation circuit C and a maliciously chosen relation circuit C and pi star such that the probability that the verifier, except for this K and the I star instance is not in the relation circuit defined by this C star is negligible.

The intuition is that the SSV is binding for some location, for some instance. So you just set the generator bar as the generator that outputs that kind of SSV. So the SSV will be binding for the I star's position. Then you cannot cheat for the I star instance. So the cheating prover can choose a relation circuit C, but it cannot cheat for the I-th star's position. That's the semi-adaptive soundness.

So to prove this soundness for this construction, the idea is that for any I star that is not in the language, you can see from the correlation intractability, essentially the idea is that if I star is not in the language, and for any cheating proof, you just switch this generator for this I star. And you can argue, if I star is not in the language, then I star over 2. Because we do the 2-to-1 trick, it will translate to a new instance at the next level, then it's also not in the L double prime.

So if the adversary doesn't cheat for the CR hash. So that's the idea. So if the query and the hash values output by the cheating prover does not satisfy the CR hash relation R, then you have this. Then if I star's position is forced to translate to another new false statement, you can use this claim to inductively argue that, initially, if you have some fixed I star, then you can track this fixed I star at each level by dividing by 2 and round up to it. And finally, you can catch it.

So in the last level of the recursion, we just send all the witness. So you must be able to catch it. So that's the idea. So that's all about the class today. Any questions?

AUDIENCE: This is the full construction get any form of, like, the [INAUDIBLE] construction?

AUDIENCE: Any sort of adaptive soundness? Or is it like a--

ZHENGZHONG JIN: Yes, so it only gets this kind of adaptive soundness.

AUDIENCE: I see. Like, some notion of this for general BARGs [INAUDIBLE].

ZHENGZHONG JIN: So for the index BARG, we can get a semi adaptive soundness and for general BARGs, I think it also has some similar notion, I think.

AUDIENCE: Is there some other construction that's adapted to sound? Or is it a very common thing to do?

ZHENGZHONG JIN: That's a good question. So in general, given any non-adaptive sound SNARG, you can always upgrade it to adaptive sound by using complexity leveraging. But you will pay due to the instance size in the security proof.

But in the BARGs, the total instance size is K times X. So it's very large. So I'm not sure you can really do that.

AUDIENCE: I see. But is it clear priori, that [INAUDIBLE] only applies to adaptively sound?

ZHENGZHONG JIN: So right, right. So there seems to be a gap, but I'm not sure how large is the gap. Maybe you can use it on clever complexity leveraging to almost close the gap, for example.