[SQUEAKING]

[RUSTLING]

[CLICKING]

YAEL T. KALAI: Let me first start with some administration. So actually, some of you noticed that the pset problem number two B, there is a slight problem. I posted a comment about this in the website.

So there's an assumption I want you to make for problem two B. Just look at the website and the announcement. If you have any questions, come to me. So that's one thing.

The other thing is in-- well, next week is Thanksgiving, so happy Thanksgiving. And the week after, I'm actually away giving some keynote talk. But actually, it worked out really well because the topic of that lecture is going to be about what's called BARGs, or batch arguments. That would lead us kind of in the final class to get our SNARGs that we were kind of building for.

And that lecture covers a work by three authors, Jain Jin, Choudhuri, Jain Jin. And Jain Jin is going to come here and present the results. So you'll get the author kind of presenting the relevant work. And then the final class, we'll kind wrap everything up and show you how to get what we worked for this entire semester.

So just of what we're going to focus on for today-- so any questions about administration, about questions from last class, anything before we go into the material? Yeah?

AUDIENCE: Yeah, when I'm doing the homework, I realized there's one thing. I'm not sure if that's something you changed with gk alpha. [INAUDIBLE] Is there any way to get subtraction to work?

YAEL T. KALAI: So let's talk. Come to me after class. OK? OK. OK. Any other questions? We'll talk about this after class.

So what's the plan for today? Or where are we kind of left off last class and where we're heading for today, so last class we defined the Fiat-Shamir paradigm. So kind of from the beginning, what did we do?

We showed this interactive protocols. And then we showed, for example, GKR, which is only for bounded depth computations. Then we showed you can build from that a PCP and then use that PCP to shrink it using cryptography and get actually four message-- That was the key in Micali protocol-- protocol, proofs, or arguments, computationally sound proofs, for all of NP where it's very succinct.

So we can take any witness for NP. And instead of sending this witness, which is long, I don't know, poly n bits, I'm going to shrink it to be an interactive protocol, so four messages, to get it-- the number of bits is proportional to the security parameter, which can be much, much smaller. So that's what we showed. That was the Kilian-Micali protocol, but that was for messages.

So now, we're kind of towards the homestretch. We want to get around these four messages and reduce it down to just one message. So how do we do that?

So last class, we defined this paradigm called the Fiat-Shamir paradigm. It was introduced exactly for this purpose to eliminate interaction from interactive protocols by essentially replacing the verifier with a hash function. And now, there was a question. It's a very nice, very simple, very efficient kind of paradigm, very elegant, in my opinion.

But then there's a question, is it sound? If you start with an interactive protocol that you can prove soundness, when you apply the Fiat-Shamir paradigm, is the protocol or the proof still sound? That's the question.

And what we proved for this protocol, Kilian-Micali protocol, is it is sound if you apply the Fiat-Shamir paradigm. But this hash function is a random oracle. So instead of being actually an explicit kind of circuit, if we think of it as a completely random function, that both parties have oracle access to, then it's sound.

But in real world, there is no kind of random oracles in the sky that all parties have access to. We have to use specific circuits to replace this hash function. And then there was this question of, when we actually use the Fiat-Shamir paradigm with a specific circuit, with a specific hash, that's computable by a specific kind of circuit, is it sound?

And that was kind of open for a long time. It was a very, very important open problem because Fiat-Shamir was used all over the place in practice. So understanding whether it's secure or not is very important. And we could only analyze it in the random oracle model, so not in real world settings.

And as I pointed out last time, the random oracle model actually worked out to be a very good proxy. So none of the schemes in practice actually that we proved security in the random oracle model turned out to be broken. So it turns out it's a very, very good proxy for security.

But still, at least for me as a theoretician, I want to understand, when we replace it with an actual function, an actual program, why is it secure. And frankly, I still don't understand why it's such a good proxy. I have to say. It's surprising to me that never, ever we had-- it's really, really approximates security very well. Even though, as I mentioned, we do have counterexamples.

We do have examples contrived, not real world examples, but contrived examples where the protocol are secure in the random oracle model and not secure when applied Fiat-Shamir And actually, interestingly, Kilian-Micali is one of them. It's actually one of the protocols where actually it was proven that, if you use a certain hash function-- collision resistant, but certain hash function-- then actually you can't prove security.

No matter which specific function you use for the Fiat-Shamir function, it will not be secure. But this specific hash function, which is collision resistant, is contrived. So we still don't have real world natural application of where security was proven in the random oracle model but turned out to be false in the real world.

So that's where we're heading to, trying to prove the security of or the soundness of the Fiat-Shamir paradigm. And along the way, what we did is I defined a certain protocol, which is zero knowledge. We kind of took a detour, talked about zero knowledge. And we looked at the specific protocols, zero knowledge protocol, for Hamiltonicity, for whether a graph has a Hamiltonian cycle or not. That's an NP complete problem.

And so here is just a recap because we'll look at it again today. So here is a proof, a three-message proof system. The prover is trying to prove to the verifier that a graph G has a Hamiltonian cycle. And we want to do it in zero knowledge, so in a proof that gives no information.

And the way we did it, we said, OK, the prover, so the prover has a Hamiltonian cycle. He does not want to reveal any information about this Hamiltonian cycle to the verifier. What would the prover do? He will choose a random permutation, pi. He will commit, put in a safe, locked safe, only the Hamiltonian cycle of pi of g.

So if c is the path Hamiltonian cycle in g, he will compute pi of the cycle, so the cycle inside pi of g. And then he's going to get a random bit. And he's going to give an answer. What is the answer? If the bit is zero, he's just going to open the cycle.

Hi. He's just going to open the cycle. Or maybe I should say pi of c. So he's going to open only the places where there's a cycle. And the guy is going to check that, yes, it's an, indeed, cycle.

But we need to make sure that the cycle has something to do with the graph itself. So if the b is 1, then we check, indeed, that you give the permutation. You do not give the permitted cycle. Otherwise, it will give information.

So you only give them the permutation. And you open only nonedges in pi of g. So now you know everywhere in pi of g that there is no edge. In this commitment, there is no edges.

So again, let me just emphasize to recap. This commitment, if n is the number of nodes in the graph, it's committing to n squared values where there's a 1 if there's an edge, 0 if there's not an edge. For any ij, it says 01 whether it's an edge or not an edge.

And if b equals 0, we only open the places where there's a Hamiltonian cycle. We check that everything there should be 1. And you check that it kind of closes, that it corresponds to a cycle. And if b equals 1, you give the entire permutation, and you open the nonedges.

Now, the claim was that this is, indeed, zero knowledge. And the intuition was, well, if you open zero, he doesn't learn anything because he just gets a random cycle. It has nothing to do with the graph. Just any random cycle, that's what he sees. He could have information about the graph.

And if he gets b equals 1, then what does he see? A random permutation and a bunch of openings of zero, that's no information. And as we said, the soundness here is only half. That's the problem.

So with the half, you can cheat. Because if you guessed-- a cheating prover, I can guess b. And if I guess b, I can at least cheat on that b. Even if g doesn't have a Hamiltonian cycle, I can always answer correctly on one of them, but not both.

So if I guess your b, I can answer. I can cheat you. So this gives only soundness half. This is just a recap. I motivated this to show that this is not even secure in the random oracle model. So actually, to get security even in the random oracle model, you need to repeat in parallel.

So today, when I say now P, V, like this, so when I say like the protocol P, V, I want you to-- throughout this class, let's think of it as parallel repetition of this with security parameter lambda copies. So now, I mean, some of the stuff is really not good for us. That's not what we want.

So think of it that we repeat this in parallel security parameter number of times. So in other words, the prover lambda number of times each time chooses a fresh permutation pi, commits. So he does this lambda times each time with a new permutation pi. This is very important. And people see why it's important that the permutation is fresh each time?

If you use the same permutation and you open b0, b1, you learn everything. So each time you give a fresh kind of random permutation, you send it lambda times, you get a b for each one. And now, you send a message for each one.

So that was the protocol. Now, we proved that it's secure in the random oracle model. Actually, we proved that any comes around protocol that has negligible soundness is secure in the random oracle model. We showed that last time. So it's secure in the random oracle model.

And now, there is a question, is it secure in the standard model when we use Fiat-Shamir? And what we're going to show today is, yes, it is secure. So that's kind of the first thing. We're going to show that this protocol, P and V-- and, again, when I say P and V throughout today's class, it's the parallel repeated version.

So we're going to show that P and V are secure when you apply Fiat-Shamir in the standard model for some commitment, not every commitment. So we'll open this box a little bit and show that this is a very natural commitment scheme. But with that commitment scheme, we can argue that this is secure when the hash function-- we'll actually construct a hash function.

I'll give you an exact hash function and say, if you use this hash function for the Fiat-Shamir, you will get security or soundness. But this hash function, to argue, you need to assume something about the hash-- I mean, this hash function assumes some cryptography. And the cryptography it assumes is the existence of a fully homomorphic encryption. I'll define what that is.

But not just a fully homomorphic encryption, we'll assume, also, that it's circular secure fully homomorphic encryption. I'll define what that is as well. I just want to say actually, today, we don't need the circular secure version. We know how to get rid of it. But what I'm going to explain today does use this version. It's just simpler.

We believe this is true. There's an instantiation based on circular secure Learning With Error, LWE. But we don't need to get into these details.

So that's what we're going to do in the first part of the class. And the second part of the class-- so now, you can say, OK, let's say we prove Fiat-Shamir. Why am I obsessed with this protocol?

And let's say we prove Fiat-Shamir for this protocol. So what? Why should you be engaged? So let me try to give you a motivation for you to stay awake and listen to me.

So why is it? So this protocol, in particular, is very important. Why? What I'm going to show is when you apply Fiat-Shamir to this protocol, you get what's called a Noninteractive Zero Knowledge proof, NIZK for short.

So once we do that for this protocol, because it was zero knowledge, we can argue that, when you apply Fiat-Shamir, you get noninteractive zero knowledge. And to get noninteractive zero knowledge based on lattices was an open question for a very, very long time.

And this was resolved in this paper that I'm going to present, in this work that I'm going to present to you by Canetti, Lombardi and Wichs. So one of the motivating examples is that we get this noninteractive zero knowledge. That's one.

But the other motivating example is that actually this technique is much more general. The technique of Fiat-Shamir that I'm going to show you is much more general than just for this protocol. Actually, it's a very general technique that can be used for many protocols.

And in particular, we used it later, this exact technique. And I'm going to try to show you how it can be used to show that the G-CARE protocol that we studied in the beginning actually is secure. When you apply Fiat-Shamir to it, you get security under learning with error, or FHE. Again, as I mentioned, this proven security can be prevented. I'm just showing you the result that uses it for simplicity.

So it's not just for this protocol to get NIZK. You can actually use it to get SNARGs. And moreover, what you'll see after Thanksgiving that Jain Jin is going to present, he's going to use this exact same tool to get batch arguments for NP.

So this kind of Fiat-Shamir is baked into all our results almost. So it's actually very general and nice. Yes.

AUDIENCE: How does that relate to-- because last time, we went a little bit over the result of [INAUDIBLE]--

YAEL T. KALAI: Great question. Great, great, great question. OK. Let me repeat your question because you have a great question. So last time, I kind of showed you that there's actually a tension. If you have an interactive proof that is zero knowledge, then I kind of gave you this high-level intuition that, if it is really zero knowledge, you cannot apply Fiat-Shamir to it. It will fail.

And here, I'm saying, oh, we're going to apply Fiat-Shamir to it. And not only that, we're going to get noninteractive zero knowledge. It's like, wait, wait, wait, wait, wait. Last time, you said you can't get your knowledge. And I'm kind of contradicting myself here.

So again, let me repeat it just to make sure we're all on board. Last time, I said a protocol that is zero knowledge cannot be Fiat-Shamir secure. Now, I'm saying, guess what? This protocol is Fiat-Shamir secure.

So first of all, indeed, I have no reason to believe that this protocol, when you repeat it in parallel, that it's zero knowledge. We know that it's zero knowledge if without parallel repetition. We know that it's zero knowledge if you repeat it sequentially. If you repeat it in parallel, we don't know that it's zero knowledge anymore. That's first.

Second, next thing you can say is, oh, so now we know that it's not zero knowledge. Almost, almost. Actually, the result does show that parallel repetition does not preserve zero knowledge. But actually, it's not going to be completely because I'm going to change the commitment in a way that you'll see in a minute, so not quite.

But then you can say, OK, but how do you get NIZK out of it? It seems completely opposite. Because I just said Fiat-Shamir in contradictory of zero knowledge. And now, I'm saying you apply Fiat-Shamir, and you get zero knowledge.

OK, so what we get is noninteractive zero knowledge. And noninteractive zero knowledge, the simulator, I didn't define what it is. So it's hard. I'll explain it as I go.

But the simulator has more power than the cheating prover. Remember, the reason I said that, if you have a zero knowledge protocol, why can't you apply Fiat-Shamir? Because we said, look, a cheating verifier can use the Fiat-Shamir function. He can behave maliciously. He's behaving like a Fiat-Shamir function. Now, you should be able to simulate the view.

So if you have an efficient simulator that simulates the view for x in the language, he's efficient simulator. He doesn't know which instances are in the language or outside the language. It's NP. It's hard languages.

So he probably will succeed in simulating the view of instances outside the language. Otherwise, he's like a P. he's a kind of distinguisher. You can use him to show that the language isn't in P, not in NP.

And that's why there cannot be a simulator. Because, the simulator, he can act like a cheating prover. In the noninteractive setting, he cannot. Because we're giving the simulator extra power to choose the common reference string. We'll talk about it when we get there. But it's a model that's a little different.

So actually, I planned I'm going to talk about this more in length when I get there. But since you asked, I want to give a preview. So indeed, there's tension. And we'll see how to overcome it.

It's actually really interesting that it's a belief that you can't use Fiat-Shamir. One, zero knowledge is a kind of in contradiction to Fiat-Shamir. But actually, we use Fiat-Shamir to get the noninteractive version of zero knowledge. But we'll go into detail when we get there.

Great question. Any other questions? So my plan next is to prove the soundness of this protocol, the parallel repeated version under Fiat-Shamir. But any question before we go there?

OK, so let's-- the result. So as I said, this is a result of a Canetti, Lombardi, and Wichs from 2019. It's so beautiful and simple. So this was an open question for a very long time.

And I want to give it to you as a kind of evidence or an example of something that we thought was so hard, so hard, so hard. And then if you just think about it like a little differently, you twist a little bit, it's so easy. So I'm going to have you guys try to come up with a proof of this.

So let's try to do it interactively. And I want you to prove to me that this protocol P, V-- but think of it, again, repeated in parallel-- is sound. So I'm going to give you a few hints, and we'll get there together. So one thing that's really nice about this protocol is I want to argue-- let's recall, what is the Fiat-Shamir paradigm?

The Fiat-Shamir paradigm says that we choose some hash function, some hash key, from-- there's some Fiat-Shamir hash function that generates a hash key. And now, we want to say, let me denote for simplicity these messages by alpha, beta and gamma. Even for now, let's even not think of this specific protocol. Let's think of just a protocol. The prover sends alpha, verifier sends beta, prover sends gamma. Let's not even go into the specific details.

I want to prove to come up with a hash function. So I want to come up with a hash function h, which generates a key for a hash and then evaluates. That's just a hash function such that this remains sound.

OK, so I have this. There's some NP language x. I want to prove whether it's in the language or not. This is my proof system. And I want to find a hash function for which it's sound.

At first, it seems like it should be very easy. Why should it be very easy? Look, take, let's say, x not in the language. We need to focus on x not in the language. x in the language, it's complete. Completeness, that's easy. I want to argue you cannot cheat.

So now, what do I need to say? I want that, for any x in the language-- so fix 1x on the language-- I want to argue that you cannot find an alpha and beta, which is h of alpha, and the gamma that's accepting. But now, this is a proof system. So every alpha has very few betas for which there even exists a gamma that's accepting.

So again, because it's fixed x on the language, first thing we know is just by definition of a proof that, for every alpha, any possible alpha, there exists very few betas for which there exists an accepting answer. So there exists few, very few, betas such that there exists gamma for which alpha, beta, gamma are accepted.

So that's the first thing we know. Because it's a proof system, by the virtue of a proof system, for every alpha, there are very, very few. Actually, note, in this protocol specifically, there's one. One.

Because if the graph g does not have a Hamiltonian cycle, take a cheating prover that sends alpha. Alpha is a bunch of commitments. Now, let's act in each commitment. Look, either they're setting a Hamiltonian cycle or not.

You can't answer both questions. We know that, if you can answer both questions, well, it won't be sound. So you can only answer one question. And each one can only answer one.

So this protocol, even for this specific protocol, let's assume even. Let's even assume what any proof tells you is that, for every alpha, there exists a few. But for simplicity, because that's the protocol we're dealing with, let's suppose there exists a unique-- this is a symbol for unique, by the way. There exists a unique beta that can be extended.

Now, we need to do is not hit that beta. So all we need-- I mean, think about it. It seems so easy. All we need is to find a hash h, like a hash key, such that even of hash key in alpha is not equal. For every alpha there exists a single bit, let me call it beta bad. That's a beta we want to avoid.

OK, so for every alpha, there exists a single beta bad of alpha. It's a function of alpha. For every alpha, for every commitment, there's a different-- either you can answer 0 or 1.

So for any alpha, there exists one challenge, beta, for which the prover can cheat. Now, the cheating prover wants to hit that bad beta because that's what he can cheat on. Our job of protecting this is to make sure that every alpha does not equal to this beta bad.

But how hard can it be? We just want to avoid one. You know what? Here, let me give a suggestion. Eval hash key or-- here, h of alpha equals beta bad of alpha plus 1.

It's not beta bad. I avoid the bad one. I'm going to use this for my hash function. Anybody see a problem with this? Yeah.

AUDIENCE: Well, it's not really unique because you're using the computational assumption for the commitments.

YAEL T. KALAI: No, actually, I'm using the computation. OK, good. So we'll say, well, actually, it's not unique because the commitment here has a computational assumption. But, actually, this commitment will be statistically binding.

So it's only the hiding, the zero knowledge, is computational. But actually, you can open it. Suppose there's really like you're binded. You can really only open it in one-- you can't open it two different ways.

AUDIENCE: But the pigeonhole principle is going to be many conditions for one.

YAEL T. KALAI: That's fine. That's fine. But you're right, there's many permutations. But suppose you can open to both. Then you can give me both this message and this message. If you give me both this and both this, it means that this g must have a cycle.

Because I know that there's a cycle. And guess what? It's not in the nonedges. So the cycle has to be in the edges. So you can open only one.

So again, all I want, it seems so easy. I have a proof system. For every alpha, actually, the prover can only cheat on one single data. All I want to do is avoid that beta. So OK, I'll just avoid that. Do beta plus 1. This is delicate. Anybody see why this doesn't work? Yeah.

AUDIENCE: Don't you have to open the commitment to find out which is the bad beta?

YAEL T. KALAI: Exactly. The problem is computing beta bad is really hard. How do you know what beta-- for example, so computing this is hard. Actually, if it were easy, you wouldn't need interaction at all. It would be NP. You'd have a succinct witness for NP.

You just give alpha. You compute. This is the beta, and then you give gamma. And everything is information theoretic. The problem is computing. The power of this interaction lies in the fact that computing what the bad beta is is hard.

Here, for example, how do you know which beta you can cheat on? Well, if you open the commitment, then you see, OK, is there a lot of 1s? Is it only in cycle? But you need to open this commitment. So computing this beta is hard without additional information.

So now, what do we do? So we can't give that as a hash function because this is a hard computation. So here is the first kind of-- so there are a few a-ha moments in this paper.

The first one is, you know what? Maybe if this commitment had some trapdoor, maybe there's some trapdoor information that, with the trapdoor, it becomes easy. So then let's do that. Let's use trapdoor in our hash function.

So now, two things I'm going to do next-- first, show you how you construct a commitment with a trapdoor. That will also be useful here, so it's going to be a stepping stone. So I'll show you specific commitment, which is what we use, which has a trapdoor to make beta easy.

And then we're going to deal with the next issue that comes up, which the next a-ha moment appears in, which is, well, you can't use this trapdoor in the clear. Because you can't give the cheating people the trapdoor. Actually, the problem will not be the cheating prover. The problem, if you give the trapdoor, it's not zero knowledge anymore.

So this trapdoor needs to stay hidden. You can't just use it. The reason it's a commitment, it has a purpose. We don't put things in the clear for a reason.

So we'll first see you can make it efficient given the trapdoor. And then we'll show how this, nevertheless, even though we need to keep this trapdoor secret, the mere fact that it exists can be useful for us to do the Fiat-Shamir.

OK, questions? Let me remind you that I really like questions. Good, bad, all of them, they're all fantastic. Yeah.

AUDIENCE: I'm just curious. So the common random string, how did you use to get step where you want to make the trapdoor somehow random enough that it's not known?

YAEL T. KALAI: So the common random string will be used in many places. The first place it's going to be used is even to make sure my commitment has a trapdoor. And then it's going to be used again and again. It's going to be used all over the place. So we'll see that next. So any other questions before we proceed?

So step one, let me just tell you how to construct the commitment so that it has a trapdoor. And the way I'm going to do it is just using encryption scheme. So let me define the notion of public key encryption. And did we define public key encryption last time?

No, right? So let me define the notion of encryption, and we'll use that as our commitment scheme. So what is an encryption? So a public key encryption with message space, there's a message space associated with it-- these are the messages that we're going to encrypt-- it consists of three algorithms.

So one is gen. It has its own key generation not to be confused with the hash function that generates hash key. This generates public key and secret key. So this just generates a pair, a public key and a secret key.

Let me just mention three PPT algorithms. So we have Probabilistic Polynomial Time algorithms. The first just generates two keys, a public key and a secret key. The encryption algorithm takes, as input, a public key and a message from the message space. And it generates a ciphertext from a ciphertext space.

And the third algorithm decrypts. So it takes a secret key and the ciphertext, and it generates a message in the message space or like that. Maybe I failed. I tried to decrypt, but I failed.

And the properties we want, the first is correctness or completeness, which just says, if you encrypt the message and then you decrypt it, you get your message back. So it just says, for every m, for any message, the probability that you decrypt-- so the probability is over a sample public key and secret key from key Gen. And then use the secret key to decrypt an encryption of m.

So it says, if you encrypt m, you generate a pair of keys, you encrypt the message m and then you decrypt it, you get back the message with probability 1. That's just correctness, what you expect. If you take a message, you encrypt it, and then you decrypt it, you should get the message back.

The important property is security or what's called semantic security. And this says that any two messages, if you encrypt one or encrypt the other, can distinguish between the two. They look the same. They're computationally indistinguishable.

So it just says, for every m0 and m1 in the message space-- so typically, we think-- by the way, the length of m is revealed. Typically, we think of the message space as having fixed size. For us, throughout this entire lecture and often is the case, we can think of m as just containing two messages, the 0 message or the 1 message. And if you want to encrypt longer message, you just encrypt bit by bit by bit. That's a common way to do encryption. Not in practice, but in theory. In practice, it's often not fast enough. But in theory, you can just encrypt each bit separately.

But in general, for any two messages, the message space, if it's just 0, 1s, for the message 0 and for the message 1, the claim is that, if you generate public key according to Gen and encrypt m0, this is indistinguishable, computationally indistinguishable, from generating a public key according to Gen and encrypting message 1. So for any messages, you know the messages. I'm telling you. I'm encrypting m0 or m1.

You can't tell the difference. You don't know. I'm giving you one of them. You have no idea if it's m0 encrypted or m1 encrypted. Note, there's no randomness over the messages for any two messages, fixed messages. Yeah. So even you don't know if I encrypt M-- if I give you two encryptions of 0 or I give you encryption of 01, can't tell the difference.

So that's the definition. And we have these public key encryption schemes and a whole slew of assumptions, lattice-based assumption, not lattice-based, like discrete log that we saw earlier in the semester and so on. But I don't want to talk about instantiation because this is more class on proof systems. And I don't want to make it too hard core crypto.

So let me just use this-- not show how to actually construct this, but we have many instantiation constructions. But suppose we have this. Let me show you what the commitment would be.

So now, what I want the commitment to be, instead of commitment, make this an encryption. Let me actually put another color. So make this be an encryption with some public key of pi.

So what I'm telling the prover now, I'm telling him, look, I want to be able in the analysis at least, to open up there and see where the 1s are and where are the 01s, so that later I'll be able to understand where the nonedges are. And so what's the bad beta?

So now, I want to tell the prover-- so please, the commitment I want you to send me is the encryption. Now, it's hiding. It's still hiding. Yeah? So it's still zero knowledge. It's still satisfies kind of-- it's locked. It's like the verifier has no idea what's in there.

And it's still binding. Because once you encrypt, you're stuck. The message is sitting there because then you can use a secret key to decrypt it. So it really has the hiding and binding property. Great, it's like a real commitment scheme. Yeah? Yeah.

AUDIENCE: Does public key encryption imply that there's no possibility of a second secret key that decrypts it to something else?

YAEL T. KALAI: OK, great question. Great question. Great. So you're asking, wait, what if the public key has many secret keys associated with it and one of them decrypts to m and the other one decrypts to m prime, let's say? So now, here I say, no.

By my definition, no. Because my correctness says that, if you choose public key and secret key pair with probability 1, you will always get your message back. So there's no secret key that will give you different message. That's my definition.

So what you're proposing is not for me. It doesn't satisfy my definition. So now, use a public encryption that satisfies my definition. Now, you're binded. Good? Great question. These are really, really great questions. Yeah.

AUDIENCE: Here in the probability expression, you're choosing public key as secret key here all coming from the same Gen function.

YAEL T. KALAI: Yeah.

AUDIENCE: But what if-- it doesn't give you anything about a different--

YAEL T. KALAI: OK, very, very good question, very good question. So the point was the following. Look, we have all these guarantees, the correctness and the security, assuming the public key is generated according to Gen.

Now, my question here, who generates this public key? How do I know it generates according to Gen? If we let the prover generate the public key, then maybe it's a bad public key that is never going to appear here. And for that bad public key, maybe your concern is valid. Maybe you can open it in different ways.

And you're completely right. That is a problem. Actually, it's not a hypothetical problem. Our scheme that we use that we want to get-- we want to get this under lattice assumption. That specific scheme we use has that property, has that problem that, if the key is generated badly, actually, there is no binding. I can cheat. I can open two different ways.

I will never be able to do that if I generate according to Gen. But if I generated maliciously, I can choose a very bad public key. Nobody will be able to tell the difference.

It will look like a good public key, but it's actually generated completely differently. And it would have never appeared in Gen. And with that public key, there is no correctness. And I can cheat and I can do whatever I want.

So very good, you found the problem in this approach. So what do we do? How do we fix it? So this comes to your question at the beginning. Well, you know what? Let's assume that there's a trusted party that generates this public key for us.

So let's assume, for now, that there's kind of a common reference string. We all agree there's a public key sitting here. That's in what's called a common reference string or a common random string. Everybody knows it, both the prover and the verifier.

We all agree on this public key. And we assume that it was trusted, like generated according to Gen. Nobody has the secret key. Nobody has a secret key. We'll never need to open the secret key.

So in this protocol, what do I do? I'm a prover. I encrypt bit by bit. So how do I encrypt? So I didn't say, but this encryption is a randomized protocol.

So I choose randomness. I encrypt the bit. I choose randomness. I encrypt the next bit. I choose randomness. I encrypt the next bit and so on and so forth.

And when I'm asked to open, I reveal my randomness. Look, how I encrypted. See, there is randomness. I took the encryption with public key and my bit 0 or 1 with this randomness. And see, that's what's written here.

So I can just open without using secret key. Nobody knows the secret key. The secret key is gone. Someone just came with the public key. And I open here not with the secret key, but I show you the randomness that I used to encrypt. And that kind of is an opening for me. Yeah.

AUDIENCE: Does that mean the CRS can't be like a fully uniformly random string? Because if you were to read out the public key, you could also pull out the secret key?

YAEL T. KALAI: OK, great, great question. The question was, does that mean that the CRS here cannot be truly random? Because it's not truly random. It's a public key. A public key may not be truly random.

And the answer is, well, first of all, some public key happened to be truly random. The public key for this scheme is not truly random. However, it's indistinguishable from being truly random.

So the scheme we actually use, the public key has the property. It's not random. It's like a LWE for those who know what learning with error is. But that's not truly random. However, it is indistinguishable from random.

Now, if you have soundness with public key that's indistinguishable from random, soundness should hold even if it's completely random. So actually, you can make this completely random. And actually, in the paper, they actually take advantage of that. Because if the public key is completely random, then you get statistical zero knowledge, computational soundness.

In this protocol, especially when you go to the noninteractive version, there's a tension. You can either get statistical soundness and computational zero knowledge or statistical zero knowledge and computational soundness. And you can play with this public key, whether to make it kind of the true public key in which case you get statistical soundness and computational zero knowledge. Or if you want to take a fake public key, which is all 0, and then you get computational soundness and statistical zero knowledge. So you can play with these things.

So this is the protocol for which we'll show that Fiat-Shamir holds. So again, the commitment is just replaced with a public key encryption. Yeah.

AUDIENCE: The common reference string, I mean, the public key is by, for example, change is a way of opening. For example, we also require the prover to provide the randomness for Gen.

YAEL T. KALAI: Oh, that's a good-- OK. OK, good. Good suggestion. What you're saying is you're saying, wait, how about he gives this public key. It can be malicious.

But when he opens, we'll ask him-- OK, so I'll tell you the problem with that. If, when he opens, he gives the randomness he used-- so you're suggesting, why don't he give the randomness he used to generate this public key? And now, you know that it's a valid public key.

That will, indeed, ensure soundness. However, that will break the zero knowledge. Because once you give the randomness, you also give the secret key away. Because this randomness was used to generate both of them.

And if the secret key, then you lost the zero knowledge. But great suggestion. OK, great, you guys. It's clear that you're thinking, which is fantastic. Great suggestions. Any other questions before we move on?

So we're making progress. Now, what do we know? So let me hide this encryption for a second. Let's go back here. Now, I want you actually to forget that this was an encryption, because we'll use another encryption.

And I don't want you to be confused with this encryption, the other encryption. What I want you to remember about the commitment, that there's some trapdoor, like a secret key. I'll call it trapdoor on purpose because we'll have another encryption which will have a secret key.

So what I want you to remember about this commitment is that there's some trapdoor information that, with that trapdoor information, you can open and know which b is bad. So a trapdoor has a commitment, has some trapdoor information, which allows you to open it, in that case, open the encryption. And then where the 1s are. And that will help you decide where the bad b lies.

Actually, you know, I think I need to also add a commitment to pi there. Because, otherwise, how do you know which-- maybe you can tell anyway. If you had a commitment to pi, it's easier. But if you don't add a commitment to pi, is it clear that just by seeing 1s it's easy to test?

Yeah, I guess. I guess if you open both, you can see both. Yeah, I don't think you need it. But anyway-- so, good. Sorry.

So now, what I want to do, I want to say here's my hash function. It has trapdoor hardwired to it. Once he knows the trapdoor, it's a trapdoor, depends on the public key and the CRS. It doesn't depend on the actual-- once he has the trapdoor, he'll know what the bad thing is and done.

So one can say, OK, use this as the Fiat-Shamir hash function. Sufficient, it's good. However, as we said, the problem is then you know the trapdoor. It's sitting there, the trapdoor. And once you're in the trapdoor, it's not zero knowledge. So that's the problem.

And then there were attempts to try to do, OK, let's obfuscate these things. And a lot of hammers were thrown in this direction. And it resulted in crazy assumptions. And anyway, there was a long line of work where that's what we tried to do.

And then here is their idea. So now, I can tell you the next a-ha moment. So again, one option is to throw what's called obfuscation at this. For those who don't know obfuscation, don't worry about it-- essentially, garbling so that the trapdoor will be really hidden there.

Nobody will be able to find it. And even then, it was a messy approach that we tried. And we got really, really strong assumptions, which I have no idea if they were true or not. It was kind of a messy approach that a bunch of papers kind of followed that approach, very complicated.

Then came this paper. And here is the beautiful idea. So the beautiful idea was the following. They said, you know what? Yeah, you can't give trapdoor. That's a problem. Give an encryption of trapdoor.

Forget about this encryption. That's a different encryption. So it's now fresh, new encryption. Why don't you give encryption-- let me denote the encryption of trapdoor here by hat. That's a common way we use fully homomorphic encryption, which I'm going to define next. So they said put in this Fiat-Shamir hash function, an encryption of the trapdoor that helps you open the commitment.

That's like, so, OK. And then what? What are we going to do with the ciphertext? We can't do anything with it. Ciphertext is like junk. What are we going to do with that?

So the idea is you know what? Don't just use any encryption. Use what's called a fully homomorphic encryption. What is a fully homomorphic encryption? I'll explain that next. But essentially, what it does, it allows you to compute unencrypted data.

So a fully homomorphic encryption is exactly the same as an encryption scheme. So a fully homomorphic, it's a public key encryption. So let me write a fully homomorphic public key encryption scheme is encryption scheme Gen encrypt and decrypt. But it also allows you to do computation on encrypted data.

So what do I mean? There is an eval function. It takes a bunch of ciphertexts, and it can do computation on the bits hiding underneath the ciphertext. So it takes a public key, a description of a circuit, c. Think of it with additions and multiplication mod 2.

And it gets a ciphertext. So encryption of b1, bit 1, let me think of it now, as I said, think of it as the message space being 0 and 1. So it gets encryption of bits. Let's say encryption-- or I'll just call it ciphertext, ciphertext 1 up to ciphertext n each one encrypting a single bit. The circuit takes as input n bits and produces a bit. And it output some ciphertext.

And this ciphertext if, here, you have encryption of b1, b2 up to bn, this ciphertext should have encryption of c of b1 up to bn. So what this eval does, it gets encryption of bits, and it can compute an encryption of the addition of the bits, an encryption of the multiplication of the bits, and so on and so forth. It can do arbitrary computation.

It can compute an arbitrary circuit c. c goes from 01 to the n to 01. It can do arbitrary computation under the FHE. So I'll just write-- maybe I'll write here the completeness property for the eval. So I wrote the completeness property where all you do is decrypt.

If you encrypt and decrypt, you get back the message. The eval also has a completeness property which essentially says-- so let me just write it down. So completeness of eval, I'll write it here.

It says that for every b1 up to bn and the probability that when you decrypt eval-- sorry, for every c, for every circuit c, when you decrypt-- so choose a public key, secret key pair.

When you decrypt, use the secret key. To decrypt the eval function, so, now, evaluate with respect to the public key and the circuit c. Evaluate on this-- and ciphertext corresponding to b1 up to bn.

So encryption public key with b1 up to encryption of public key and bit bn, so encrypt each of these bits. You'll get encryption of b1 up to encryption bn. Then evaluate. Do homomorphic computations on these encrypted bits with respect to circuit c.

Now, you get a ciphertext. You should get a ciphertext that encrypts c of b1 up to bn. And indeed, the probability that when you decrypt you get c of b1 up to bn is 1 or close to 1.

So again, what eval does, it does a computation on this encrypted data. It computes an arbitrary circuit c that, when you decrypt it-- and did you get c with probability very, very close to 1. I wrote here 1 minus negligible as opposed to 1 because our scheme have this negative probability of error. So that's why I--

So again, high level what the FHE, Fully Homomorphic Encryption, allows you to do is to do this computation on encrypted data. Questions about the fully homomorphic encryption? We're good? Yeah.

AUDIENCE: I guess does it matter what c is? Oh, never mind. Never mind. I was thinking just arithmetic or--

YAEL T. KALAI: Good, good. No, that's actually a very good question. It's a very good question. So the question was this circuit c, what is it? So the circuit c, think of it as a circuit that has addition and multiplication gates mod 2, so any polynomial size circuit that has addition and multiplication gates mod 2. Great, thank you. Other questions?

So now, let's go back. So now, we have an encryption scheme that we can do computation and encrypted data. Now, let's go back.

Remember, where are we? We're saying all we want is to avoid this bad. We want a hash function, its only purpose to avoid one single point. How hard can that be?

I'll tell you the hard problem is it needs to avoid the bad point for each and every alpha. That's kind of the hard part that, for every alpha, you need to avoid the bad one. If you just want to avoid the bad one for one alpha, choose random. Probably, you're not going to hit the bad one. But you need to avoid the bad one for each and every alpha. That's kind of harder.

But if we have the trapdoor, then it's easy. But we said, look, we can't give the trapdoor. That will break zero knowledge.

So instead, we say, you know what? Let's give a homomorphic encryption of the trapdoor. Now, this seems not helpful. Why is it not helpful?

OK, let's say you can compute on encrypted data. You do the computation of computing bad. So now, here's my hash function. It has trapdoor, hardwire. So think of it like h of, instead of trapdoor, it will have hat of trapdoor. Why not? And it will do this computation under the hood with eval under the hood of the encryption, very nice. So now, on input alpha, you won't get the bad alpha plus 1, very nice, but encrypted. So why is that helpful? Yeah.

AUDIENCE: [INAUDIBLE] you could also encrypt the eval part, so that encryption, the ciphertext, the two different messages are now equal.

YAEL T. KALAI: Again, what are you saying? Encrypt?

AUDIENCE: Encrypt the eval-- the left-hand side of the equality, the eval hash [INAUDIBLE].

YAEL T. KALAI: Oh, you can encrypt-- exactly. So that's kind of what I'm doing. This is the shorthand for-- this is actually eval of-- this is kind of the hash key. The hash key will have trapdoor in it. And an input alpha, it should output this.

And what we're doing is we're saying, OK encrypt-- or you can think of it the hash key being trapdoor and encrypt the entire thing. You can do everything under the hood of the encryption. You want the output to be different than beta. You get a ciphertext.

I want this to be different than beta. That's what I want. I don't want underneath the hood to be something that's different than beta. So this seems like a bad idea. It gets us nowhere.

Yes, again, where are we? We said, if we knew trapdoor, we would be done. The hash function, the eval of the hash function, will take this trapdoor, compute beta, and output not beta, move it by one, move one of the bits by one, whatever, as long as it doesn't-- it's not the beta.

And now, they say, no, let's do everything under the hood. What? Then we'll get an encryption. Why would that not be-- it seems like you're not getting anywhere to me. It seemed like, when they first presented this idea, I'm like, OK, this seems completely useless. So what? Now, everything is under the hood. And guess what? We're almost done. Yeah.

AUDIENCE: [INAUDIBLE]

YAEL T. KALAI: You take the randomness.

AUDIENCE: Define the randomness so that computes encryption of eval plus 1 gets us that--

YAEL T. KALAI: So you know what? I want to say something even before that. There's even a mismatch. Beta, let's say, consists of lambda bits. For example, in our case, because we repeat it, its lambda bits.

This is not even lambda bits. It's much more than lambda bits because we encrypt bit by bit. It's much bigger. It seems like completely bogus.

This is going to be the length of beta times security parameter. So it's not it doesn't even correspond to-- the sizes don't even match. So it seems like an encryption is not helpful. Yeah.

AUDIENCE: How could it computer beta bad given alpha has trapdoor?

YAEL T. KALAI: Oh, Good The only way he can compute beta bad is if he has the-- oh, how can you compute given the trapdoor, you're saying?

AUDIENCE: Yeah.

YAEL T. KALAI: OK.

AUDIENCE: Then it will [INAUDIBLE]?

YAEL T. KALAI: Yeah. So what he will do? The trapdoor in this protocol allows you to open the commitment. So now, you see each point if it has a 01, 01, 01. So now, why do you know what's beta bad? Because, now, you look.

If it has just a Hamiltonian cycle, I mean, the truth is, it's easier to see it if you have the pi. Suppose you always also committed to pi. Well, it's just easier to see it that way. Suppose I tell you, don't only commit to that. Commit also to your pi.

Now, you see what's in there. Either if you remove pi-- so you look at all the edges of pi of g. And you make sure that these edges are 0. Everywhere, they're 0. If one of them is 1, you know that he won't succeed.

Then this is not bad. Then the b equals 1 is not bad because he will fail. The bad is where he will succeed. A bad beta is the beta that will allow him to succeed.

So now, if you can open everything, now you can see. If there is one edge of pi of g, if there is a nonedge that has a 1, then you know this is definitely not bad. And so he won't be able to see-- so, this, this is the beta bad.

On the other hand, if all of the nonedges are, indeed, 0, you know he can't open Hamiltonian cycle because there is no Hamiltonian cycle. So then this becomes the bad. So if you know what's in the commitment, you can tell which one is the bad. So the trapdoor allows you to compute the bad efficiently. That's the point. Yeah?

AUDIENCE: So does that mean the [INAUDIBLE] the random coins used in each [INAUDIBLE]?

YAEL T. KALAI: The random coins used?

AUDIENCE: The randomness that's used inside each-- is that what the trapdoor is?

YAEL T. KALAI: No. I'm thinking of the trapdoor as being the secret key corresponding to the public key. I'll tell you the problem. The problem of using the randomness, then it's chosen by the cheating prover. I don't know how he's going to use it.

The trapdoor, you write the trapdoor-- this is an important point. So note, I can open this commitment by giving you the randomness that I used to generate this commitment. That would be a good trapdoor. But when I designed the Fiat-Shamir hash function, I have no idea what randomness the cheating prover is going to use. Maybe it's even deterministic. I don't know what he's doing. He may just give me-- I don't know. So I don't know ahead of time it's random coins. And different cheating provers will have different random coins.

So I want to put something in the CRS, in the hash function, the Fiat-Shamir hash function. So this trapdoor is actually going to be the secret key corresponding to the public key. The secret key is not going to be used in here. It's never going to be used in the protocol, but it will be used in the analysis. OK? Great. Yes.

AUDIENCE: Is this only if you do an interactive proof where you commit to the witness, or can you draw this to put first [INAUDIBLE]?

YAEL T. KALAI: Again, you're saying, is this--

AUDIENCE: To proof system where you commit to the witness, or can you draw this to proof system where the first [INAUDIBLE]?

YAEL T. KALAI: So this technique works for this protocol where you commit to something in open. And after, I'll show you why. I'll show you that it's actually more general. And it applies to a large, actually, slew of protocols where there's no commitment whatsoever.

For example, it applies to GCR protocol where it's just sum-check. It applies to sum-check protocol. And so it's not specific to this.

And I'll show you later all you need is this trapdoor that tells you where the bad kind of challenges are. And once you have a trapdoor that tells you where the bad challenges are, then you can use it. But we'll mention that at the end of class, like towards the end. But yeah, we'll see that it's general.

So in some sense, you don't even need to look too, too detailed into that protocol. The idea here is more high level. We say, look, in an interactive proof, there's the beta that are bad, bad for security, that are good for the adversary. But they're bad for security.

If the adversary managed to hit a bad beta and alpha for which h of alpha gives him a bad beta, then he can cheat me. I want to avoid these bad betas. And now, we're saying, well, there's only one bad beta for simplicity. This one has one bad beta, so we're good. If it has one bad beta, I want to avoid that one.

Now, if there's some trapdoor that allows me to compute that bad beta, then I can't put the trapdoor in the hash function and give it to the world. There's a reason why it's a trapdoor and not public. Here, in this case, it's for zero knowledge. But what I can do is give an encryption of the trapdoor.

And actually, in the actual scheme, it won't be encryption on that trapdoor. It'll be encryption of zero. Nobody's going to ever decrypt this thing. But let's think of it as an encryption of the trapdoor for now.

Now, again, where are we? Why did we make any progress here? So next, is the most amazing point, I think. When I saw this, it took me a very long time to process because it's such an ingenious idea.

But here's the idea. So they say, yeah, you know what? Nevertheless, my hash function-- so then they say, OK, you're right. This kind of hash function that has the trapdoor in it and you evaluate the trapdoor, you're right. It's not good. It doesn't even type check. The output is not the length it's supposed to be. Yeah, this is nonsense. Let us fix this.

And here's how they fix it. Let me put the eval here because it'd be nice to see. So here's their fix. They said, you know what? Fine, that may have been the wrong function. We're going to encrypt. So our CRR hash key is going to consist of encryption of another function, g.

By the way, I'm using the hat to denote fully homomorphic encryption as opposed to standard encryption. That's often we use in the literature. The hat is kind of a shortcut to denote public key comma encryption of g. And when you use this hat, we usually use it when we want to talk about fully homomorphic encryption.

So now, they say, this is going to be my-- nevertheless, I'm going to encrypt a g. What's this g? We'll see. I'm going to encrypt a g. So here's their hash key. The hash key is going to be I'm going to generate a public key and encrypt a function g.

So it's like the public key corresponding to g and then g, which is just another shortcut to say encryption of g. And when I say encrypted g, just think of the description of the circuit. So there is some way to describe the circuit. I encrypt the description.

So I encrypt this description of a circuit g. What this g is we'll see. But that's my hash key. So this is the hash key. How do you evaluate hash key and alpha?

They say you know what? Just compute g of alpha under the hood. How do you compute g of alpha under the hood. Essentially, they say, do they eval of the f and g. There's two evals, so it's a bit confusing. Because we usually denote the evaluation computing the hash function by eval. But now, we also have the eval of the full homomorphic encryption scheme.

So take the fully homomorphic encryption scheme. Take the ciphertext g. So take the public key corresponding to the ciphertext. What's the function? What's the circuit c? It's going to be kind of a universal circuit sub alpha. I'll tell you what this is in a second. And the ciphertext is g.

And this universal circuit of alpha, all it does, the definition, it takes as input a circuit g and it outputs g of alpha. So essentially, what we do, what this hash function does, is exactly what we said before. It takes an encryption of g. How does it relate to the trapdoor? I'll tell you in a second.

Before, we said, g, it just has the trapdoor in it. Before, we say g is just this g, computes bad. And it has the trapdoor. And we encrypt g.

Now, they say, OK, you're right. There's a problem. We'll tell you what g is in a second. But same idea, same idea, we encrypt this g. Maybe it'll be a little different, but we encrypt g. And the hash function computes g and alpha under the hood.

So it takes the original first message alpha and computes the function g under the hood. We just went over for it. It doesn't work. Then you get the encryption. It doesn't even type check. What?

So here's their idea. Now, this is the most beautiful part of today's lecture. Let me just write it short kind of hand-way. This is going to be g of alpha encrypted because this is what u does. u does, it computes g of alpha. So what you get back is encryption of g of alpha.

Now, they say, OK. And by the way, let me first say, in their scheme, what is g? It doesn't matter because it's encrypted. It's the all-zero circuit of a certain size. So in their scheme, this g is going to be all-zero circuit of size T0. What T is we'll see in a second.

But that's all it is. This is the Fiat-Shamir hash function. The question is, why is it secure? Because it seems like there is no way this works.

But let me just say, again, the Fiat-Shamir hash function, what does it do? It generates a public key. It encrypts in the actual scheme, all-zero string, 000 up to T times. We'll see what T is in a second. In the analysis, we're going to play with this g. But this is what it is.

And when you give him alpha, it computes the circuit, which is the all-zero circuit, under the hood. That's all it does. And that's the b. That's going to be the beta. You should be very confused, but I'll explain. Yeah.

AUDIENCE: All-zero circuit board?

YAEL T. KALAI: Just think of the zero-- everything is, I don't know, whatever. It doesn't matter, all pluses, all plus gates. Or it actually doesn't matter, because nobody's going to ever decrypt it.

So I guess I'm saying, each circuit, you can describe it with bit 010101 in some way. So put all 0 bits. So I don't know if it even describes a circuit. Yeah, Anand.

AUDIENCE: Could you just use the original beta bad plus 1/2 [INAUDIBLE], the prover has some-- because gamma is a function of beta. So just do that under the FHE and get an encryption of gamma, or is this not what [INAUDIBLE]?

YAEL T. KALAI: No, I'll tell you why it doesn't work. OK, good. No, great question. So the question was the following. You're saying you know what? Change the protocol altogether. Change this protocol and ask the guy, you don't need to be blinded to this Fiat-Shamir paradigm. Change it a little bit.

And now, you put an encryption of the trapdoor. So now, he computes better under the hood. Tell him to compute gamma also under the hood. And then we don't have a problem. The problem is that he can cheat. Why? Because when he answers gamma, he knows the trapdoor.

No. But actually, one second, you're right. Under the hood, he knows the trapdoor. Yeah. So hold on. Yeah, the problem is that, under the hood, he knows the trapdoor. However, I'm not sure actually that that's a problem.

Because at that point, let me think for a second. I think it actually may work. So look, it changes the protocol. For example, now, it's no longer publicly verifiable.

Now, his message is encrypted, so you need to decrypt in order to learn it. So there are issue with this. But also, I'll tell you more than that.

If, indeed, you put in the Fiat-Shamir in the actual scheme-- you should think of it, the people who generate Fiat-Shamir, they actually don't know the trapdoor. They don't know the trapdoor. There's a scheme, and it's sound. And there is a trapdoor, but nobody knows it. And the people, they don't actually know the trapdoor.

So in the actual scheme, there won't be any encryption of the trapdoor. The idea is we'll say, well, actually, we don't use the trapdoor. But it's indistinguishable. You don't know if you're encrypted zero, encrypted the trapdoor. So we're going to prove soundness, assuming you encrypt the trapdoor.

But in the scheme, won't encrypt the trapdoor. So then in the scheme, the gamma is going to be useless because it won't correspond. Yeah? OK, great. OK, great, great, great questions.

So where are we? So we said, here's the Fiat-Shamir hash function. The Fiat-Shamir hash function will take actually an arbitrary g of a certain size. It doesn't matter because nobody's going to decrypt that ciphertext anyway. And all it will do is a computation under the hood.

And now, it seems like I don't know it seems like that cannot work. Nevertheless, I'm going to prove security in, like, two lines. And the proof is ingenious, really.

This construction, it's very tempting to try. It just doesn't work. When you look at it, it's like OK, and then what? And then what?

But here's their idea. Their idea is following. Look, all we need to argue is that, for every alpha, g of alpha is not beta bad. That's all we need.

So then in the analysis, I'm going to say-- so again, this is the actual hash function. You have an encryption here of g. What g? This g is-- I don't care. It's like the fixed circuit, whatever circuit you want to put. And nobody's going to decrypt it. In the analysis, I'm going to say, you know what? I'm going to replace this g with a very special g, with a g whose encryption is not beta bad.

But you're like, which g hat? Which g hat? How do we construct the g so that we know that encryption of g and alpha is not beta? Yeah?

AUDIENCE: Just a syntactic question, what does this notation mean? Is it like an encryption of g-- is the hat over the whole thing?

YAEL T. KALAI: Yeah. Sorry, yeah. Sorry. This is an encryption with public key of g alpha.

AUDIENCE: Yeah.

YAEL T. KALAI: Yeah, sorry. Thanks. Thank you. So any idea what-- OK, I'll tell you. You won't believe me, but I'll tell you anyway.

It's math. So of course, you'll believe. But you'll be mind-boggled for a while. So here's their idea. Their idea is they say, OK, here's my g. I need to ensure that you're not beta, so I'm going to make my g be the decryption secret key of beta.

And I'm just going to add the 1 to it so I know you're not-- I guess we're right XOR because we're over 01. I'll XOR with this with 1. That's my g in the analysis.

So in the analysis, I'm going to say, suppose you succeed in breaking the Fiat-Shamir. You're also going to succeed with this g. Because it's encrypted, you don't know the difference. You have no idea what's underneath the encryption. So you should also succeed in breaking with this g.

Now, this g looks bizarre because I'm decrypting. This is not a ciphertext. This is just a bunch of strings, a 01 string. It's a random string. It's not ciphertext. What does it mean to decrypt this thing. It's like, this makes no sense.

But if the length of this is the-- take a ciphertext that the length is like these lambda bits. Look, this is a function. So you can compute this function, not on a ciphertext, but compute this function.

So I decrypt the beta message, which is not even a ciphertext. And I add 1. I XORed with 1. I XORed with anything that's not 0 just to move it from the bad.

Now, I want to argue, now, we know that encryption of this cannot be beta. Why? Now, I can argue this. Why? Because if it was equal, if you decrypt this, then you'll get a description of this. If you have two strings and they're equal, if you decrypt one, you can apply decryption, but they'll still be equal. The decryption is just a deterministic function.

So I guess. For this g, I want to argue that this, which is Fiat-Shamir hash function in alpha-- this is the Fiat-Shamir hash function in alpha. I want to argue this. I want to argue, if it was beta, I'm going to get a contradiction. I want to argue it's not beta. Why?

If it was, then I can apply decryption on both sides. Then it means that decryption of this will be equal to decryption of this. Of course. What is decryption of this? g alpha. So I just decrypt the encryption of g alpha. I get back g alpha.

But I know g alpha is not the decryption of bad alpha. Because g alpha is the decryption-- not plus 1. So it can't be.

So again, in the analysis-- so let me maybe write this a bit formally. So let me write it up there. I'll use this part. So here is the actual proof of soundness.

So suppose-- oh, sorry. There's one thing I glossed over fast. We'll go over it in the analysis. We'll see it.

So I say, suppose there exists poly size cheating prover. And he fixes some graph g that's not-- does not have a Hamiltonian cycle and cheats and cheats with probability big. Let's say greater than epsilon. Epsilon is non-negligible.

Then I'm going to say, then I want to argue. Then it means that p star also succeeds with probability close to-- epsilon maybe minus negligible, but similar probability. If I change the hash key-- before, the hash key was public key and some encryption of some bogus g.

Now, I say it will also succeed if I take public key and this specific g. Let me call this g, g star. So that's the g in the analysis. So again, how does the analysis go? I say, suppose there exists a cheating prover that takes a false statement and succeeds in proving nevertheless and convincing.

Then I want to say that he will also succeed with hash key which is public key and encryption of g star, of this g star. Why? By semantic security because you can't distinguish between encryption of g star and encryption of g. Yeah.

AUDIENCE: So here, we're using the proof is efficient?

YAEL T. KALAI: We're using the fact that the proof is efficient.

AUDIENCE: So there's computational soundness.

YAEL T. KALAI: Computational soundness. Computational soundness, yes. We're using the fact that the poly size is efficient. And therefore, he cannot break semantic security. So he cannot distinguish between g, whatever the bogus g that we put in the zero's one, the all-zero circuit or whatever, and this g star.

I cheated slightly here. But let me move on, and we'll go to the cheat in a second. But let's say you believe that even though you shouldn't. But it's very delicate, so we'll go there.

OK, semantic security. Now, I'm saying with this g star, now, information theoretically, he cannot cheat. He's stuck. Because guess what? You're saying he produces alpha, beta, gamma accepting, right? That's what he means he cheated.

It means that he got alpha such that h of alpha is bad. Because you can only cheat with a bad alpha. That's the definition of a bad beta. That's the definition of a bad beta, one that you can cheat with.

So we found an alpha so that h of alpha is bad. But this g star never hits bad. That's how we made it. It's so that g of alpha is never the bad alpha. Yeah.

AUDIENCE: So g start is [INAUDIBLE].

YAEL T. KALAI: Ah, that's what I glossed over, exactly. Good. So that's the proof. However, there is a cheat. And the cheat is, look, g star, to compute g star-- I said, just describe g star. When I mean encrypt g star, I mean give the description, like encrypt the description of g star.

So let's recall what does g star do. g star decrypts. So you need to know the secret key corresponding to public key. We said that you can encrypt-- semantic security holds if, for any fixed messages m0 and m1, you cannot distinguish between-- is it here? Yeah. For any fixed m0 and m1, you cannot distinguish between encryption of m0 encryption of m1.

But here, I'm saying you cannot distinguish between encryption of the secret key and encryption of, let's say, other bits. That's a different story. And indeed, we assume what's called circular secure FHE. We assume all that you cannot distinguish an encryption of the secret key an encryption of the all-zeros. That's an assumption. You have a question? Yeah.

AUDIENCE: So far it seems like b beta, beta bad, is also given, not computed? How do we--

YAEL T. KALAI: Oh, no. No, it's computed. Sorry, sorry, sorry. OK, good. No, what I meant here-- OK, thanks. Thank you very much for your question. Here's what g star alpha does.

g star of alpha, it has-- actually, let me write more carefully. It has trapdoor and secret key hardwire. The first thing it does, it computes bad alpha using trapdoor. This is kind corresponding to our scheme.

Then he takes what he got, the bad alpha, he decrypted using the secret key. Then he XORs 1 to the least significant bit or something like that. So yeah, it does compute it. Yeah, so this g star, you need to encrypt trapdoor. And you need to encrypt secret key.

Now, encrypting the trapdoor is not a problem. It has nothing to do with the key. But the problem is you also need to encrypt the secret key corresponding to the public key. But yeah, thank you for the question. That's a great question.

So g alpha-- I want to repeat. g star alpha, what does it do? It takes alpha. It first computes the bad beta corresponding to the alpha using the trapdoor. Then he takes the string. He decrypts this weird string using the secret key. And then he XORs the least significant bit with 1 or any bit with 1 just to make it not equal.

AUDIENCE: What is trapdoor? Is it still the secret key?

YAEL T. KALAI: It's the secret key corresponding to this commitment. That's why I said this trapdoor, in some sense, is also a secret key of a public key encryption. But there's two encryption going on. One is the FHE encryption. And one is how we implement this commitment.

So you write that we implement this commitment also with an encryption scheme, but it's better to forget it. Because, otherwise, it's confusing that there's two encryption scheme. So here, I want to say, suppose you have a commitment. And there's a chapter that allows you to open it. Forget that this trapdoor is also a secret key.

So now, I'm saying you need this trapdoor to open the commitment, which you're right. This trapdoor happens also to be a secret key, too. Because that's how we implemented the commitment. And it has a secret key corresponding to this same public key. You see?

So I guess what's the problematic part is that the hash key consists of a public key of a fully homomorphic encryption scheme. And to describe g star, you need the trapdoor, which is not a problem. You could just encrypt this trapdoor under public key. It's a fresh public key. You can encrypt any-- this is kind of fixed compared to this public key.

But you also need to encrypt the secret key corresponding to this public key. And for that, we need to rely on circular security. So let me just quickly write what I mean by circular security, and then we'll break. So I'll write it here.

It just means that public key and encryption of the secret key is indistinguishable from public key and encryption of, let's say, the all-zeros. And let me just mention that secret key here you can think of it as lambda bits. Let's say a security parameter number of bits.

Then if we think of bit-wise encryption, it's like encryption of secret key, the first bit of the secret key encryption, the second bit, it's like lambda encryptions. And this also, I mean, encryption of 0, encryption of 0, encryption of 0. We do everything bit by bit.

And the assumption is that you have an FHE scheme so that you cannot distinguish between whether the lambda encryptions-- each one encrypts a bit-- whether these encryptions are all-zero or whether they correspond to the bits of the secret key. So that's an assumption.

By the way, so the question is, well, do we have FHE that have circular security, question mark? Well, we do under an assumption. It's called circular secure learning with error. So we just move this circular secure into the assumption.

We believe this assumption to be true. We don't know. But actually, we can remove all this. It turns out we don't need to put the secret key there. And there's a follow-up work that did more work, a lot of lattice work.

Here, I didn't talk at all about-- I mean, as long as you have fully homomorphic encryption that's circular secure, we're all set. I don't care what this is like. But then there's follow-up work that looks at the specific fully homomorphic encryption scheme based on lattices and does a bunch of lattice tricks to remove the need for circular security. So this is something I didn't show you, but there is a follow-up work that does that. So I want to take a break. But before a break, are there any questions? Yeah.

AUDIENCE: So here, the size of the circuit T has to be large enough to contain this--

YAEL T. KALAI: Good. Exactly. So good question, very good question. Guys, you're really having fantastic questions today. Thank you. So going back, what is this T? This T is the number of bits needed to represent this g star.

Because in the analysis, I convert the 0 to the T with g star, with the decryption of g star. So this is exactly what T is in the analysis. Any further questions?

So let's take like, maybe, I don't know, 8 minute break. It feels intense. Take like 8 minute break or so. And when we go back, we'll do the NIZK.