TADGE DRYJA: And that brings us to today's lecture, which is also something I was not as familiar with. So sort of a disclaimer-- I might have gotten little things wrong, or it might not be quite as clear. But do ask questions.

So if it's stuff like Lightning Network or Discreet Log Contracts, well, for me-- that's sort of easy, because I know it really well.

This stuff is things I've known about, but I've never really went as into the details. So it's a little bit like, OK, figure it out well enough to explain it rather than just well enough to understand how it works.

So the basic idea is you want to hide the output amounts. So we saw last week how looking at output amounts can deanonymize and let you link a lot of different things. And so we'll look at how to do that.

How do we hide output amounts? And so we'll talk about commitments. So first, we'll talk about the reasons why and how to do it. Then we'll talk about the definition of commitments. What is a commitment in cryptography?

And then we'll talk about Pedersen commitments, and then we'll talk about range proofs and put it together to get what's called "confidential transactions," which is a vaguely confidential transaction-- it just means the output amounts are secret.

So interrupt with any questions at any point of the way if something's not clear. So coinjoin-- last time we talked about combining transactions with other people where you have your input-- they have their input. You have your output-- they have their output.

And if you shuffle the ordering of the inputs and outputs, it might not be clear who's sending what where. You're aggregating it, and in the process, you're losing information, because no one gets to see the border lines between-- hey, this is one transaction-- this is another

transaction-- it's all in the same transaction, so you can't really tell.

But one issue-- and a really big issue-- was that the output amounts can often reveal who's sending what where. So in this example-- well, maybe I don't know who's A, who's B, who's C, who's D-- these all look like uniformly, random addresses or pubkeys.

But if I see an input with 10 coins-- an input with 2 coins, and then an output with 2 coins, and an output with 10 coins-- well, there's probably something like this going on. Why else would you do this?

If you were trying to send 2 coins somewhere, why even involve this 10-coin input? Why not just say 2 coins to 2 coins, and you're done?

So if the goal is to obscure the transaction graph and have better anonymity because no one can tell what's going where-- well, this doesn't really work. I think it's going like that.

And then I showed that there is a way to do it, whereas if you have 2 to 8, then the 8 is obviously from the person with 10. But the 2 different 2 coin outputs are unlinkable. So that helps, but it's not great.

And then for things like the aggregate signatures, maybe you can get people to aggregate their transactions together because it's cheaper and smaller.

But in general, they'll have totally different input sizes, totally different output sizes, and it won't really give you any anonymity. And it won't give you any privacy, and it hurts fungibility. So how can we do this?

Wouldn't it be great if we could hide the output amounts? That would be really cool. So instead of having it like this-- it's just you don't see how many coins there are. Why don't we just delete the output amounts entirely? Why even that saves 8 bytes.

So we can't just delete it. So what does this mean? This would be really cool, though, if we could somehow obscure this so that no one can tell what the amounts are.

And, I guess, initially, so they're going to be some transition since Bitcoin wasn't built with this from the get-go.

At some point, there will be a transaction where you can see the input amounts, but then you can't see the output amounts. But then you can have bounds and say, well, neither of these

can be more than 12.

And then as it goes forward, and if the transaction graph gets bigger, those bounds start getting really wide because you're like, well, this might have been 11, and this might have been 1, and then the next transaction if it's mixed in with a bunch of things that have a pretty wide range, you have to take the max and min of all those things. And, the min is always going to be 0, so it works pretty well even with this transaction.

You could also think of a new system where either you start with obscured amounts, or the first mining amounts are non-obscured, and then you mix them.

Anyway, so this seems like it'd be really useful because it would allow you to combine transactions and get better privacy. Also, just really nice in general, even if you're not trying to combine with anyone else-- it's really cool that people can't see how much money you have.

If people can see how many coins you have, they can try to charge you more-- they can try to rob you-- there's all sorts of reasons why people don't usually say how much money they're carrying around, or how much they make, or how-- things like that.

People are sensitive about pricing information, and some of it's societal taboo, but also a lot of it's like I don't want people to know how much money I have.

And I think there's-- I've read recently like Uber doing this thing where they like, oh, you're rich-- we're going to charge you more. And people don't like that. It has been historically doable. Companies segment things, but often the way they do it is they're like, OK, well, we're going to make first-class seats and then economy seats. And the first class seats don't take up the space of 10 economy seats-- they take up the space of maybe 2, but they cost like 10x.

And the idea is, well, if you're rich-- we're going to charge you more. You can have the fancy stuff, and we're going to make more profit off of that. That's more OK, I guess, but if it's just-- no, you're getting the same economy seat, but we know you're rich, so we're going to charge you more-- that-- I don't know.

And we're getting into that future where there's enough data that a lot of companies can do this. And some people, probably myself included, is like wait, no, I don't like that. I like the way it was before where companies didn't really know that much about me.

So this is an effort to try to keep things obscured-- try to keep the customer a bit more

anonymous with respect to merchants so that they don't have as much information to charge more.

And then-- rob you-- that's maybe more Bitcoin-specific where like-- yeah, I mean if you're walking around in an area that's sort of high-crime with a bunch of money hanging out of your pocket-- that's a bad idea.

The internet is sort of a high-crime area, especially, with respect to Bitcoin. And it's really easy to see how much people have. And so people-- the hackers know who to target, especially if it's like an exchange-- they're pretty clear-- OK, this exchange has tons of money.

So it might not help the exchanges that much-- it obviously has a lot of money, but anyway. So this would be really cool to do. I think people sort of agree-- this would be really cool. This is a nice thing to have.

And it also helps with privacy. You're trying to make it hard to link different outputs to each other or link the outputs to a specific person. And hiding the amounts makes them really hard to distinguish.

So if you don't know the amounts, they become completely uniform where this is a completely random address. I have no idea how many coins there are. I have very little way to distinguish between these outputs. I can try to trace the graph and say, OK, well, this came from here-- this came from here, but they all look the same.

So I've got this big graph with no real information other than the links between the notes, which is still information. But it's much harder to tell what's going on.

So we're sold-- how do we do it? First, we need to define-- what exactly are we trying to do? What are we hiding from whom? Who knows what and when kind of thing?

So the long-term state of the system-- no one can see any-- no one can see it in the amounts. A has signed a signature from key A-- a signature from key B-- some amount of coins. You don't see it. you don't see any of this.

So that's the long-term state. But wait, people receiving the payments should probably know how much they're receiving, right? And they should probably know how much they have. If you don't know how much money you're getting-- that's not a very useful payment system.

People sending should-- they should also know how much money they're sending. If the amount can be completely determined by the receiver-- that's also not a good payment system. I'm just going to take lots of money.

So we can have two views. We can say, OK, only the sender and receiver know the amounts. How about that-- where from the network's point of view-- so if I'm just the third party looking at this transaction-- I don't see any of the amounts. I just see there's no-- I have no idea how many coins are being sent.

But from the sender's view, the sender says, OK, I know I have 2 coins here. 7 coins-- I changed it from 2 and 10 on that line-- 2 coins here, 7 coins here. And I'm sending 7 coins here and 2 coins there.

So the sender knows the whole thing because they're the ones creating this transaction. And then maybe the receiver doesn't see everything. Maybe the sender can prove to the receiver a single amount but hide everything else. That would be ideal.

So the receiver only sees, oh, I'm receiving 2 coins, but I don't know. If I know this, then I can figure out what these are, or at least the sum of them.

But if I don't know any of these things-- I don't know how much the person sending me money has. I just know, oh, I got 2 coins. Cool, that's what I wanted. But I don't learn anything about their inputs or their other outputs within the transaction. So that would be ideal if we can do that.

So what we want is a system where network doesn't see any of the amounts. The sender presumably sees all of it. Now, there could be cases where there's multiple senders, and there are two people working together to make a transaction, and maybe we don't see each other's inputs, but we'll keep it simple for now where there's a single sender.

And then the receiver only sees their own output. So that would be a really powerful system. Then the receiver would only know they're getting the right amount and not learn anything else about the sender's coins.

So we might want to-- like I was saying, you might want to hide per output. So if you have a-- if it's a global-- either you see everything or you see-- sorry-- you see nothing, or you see everything-- that's still pretty good. That's still better than what we have when everyone sees everything, but it is better if you can hide the rest of these three things from the receiver.

So any ideas? Some kind of encryption? Hide the amounts so the only people with the right private key can see the numbers?

So, I don't think I've even talked about encryption ever in this class, because there is no encryption in blockchains. [LAUGHS] Which is something you-- if you read something about blockchain and you're like blockchain is encrypted. And you're like no, no-- they don't know what they're talking about.

So I haven't defined encryption. But encryption is-- generally, you have some kind of private key-- you take a clear text, you apply-- you do some-- so like I had with signatures, you have a set of functions.

So encryption is-- you've got some encryption function which takes a key and a message and outputs a ciphertext. And then you've got some kind of decryption function, which takes off in the same key or something and then the ciphertext and outputs the same message.

And this is symmetric encryption because you're using the same key. You can also have asymmetric encryption where this could be a public key, and this could be a private key.

So in order to decrypt, you need the private key, but in order to encrypt, you only need someone's public key.

So, in practice, all the things that people-- in practice, you usually stage it, so you have an asymmetric algorithm to agree on a key, and then you just use the symmetric encryption and decryption where the keys are the same because it's way faster.

So you can do this operation on the order of hundreds of megabytes a second, whereas the asymmetric ones are slow-- it's sort of like signing. Maybe I'll go into it later. But if you understand all the signing and stuff, it's not too bad.

But so maybe you could say, OK, well, there's some kind of private key that the sender creates-- encrypts the amounts, and then they put the encrypted amounts in the transaction, and they reveal the key to the receiver. And, potentially, you could have different keys for each amount-- that would work. Any problems with that idea? You can encrypt everything. Yeah, you know?

AUDIENCE: You would need to send over a public key or something so they can decrypt it.

TADGE DRYJA: Yeah, oh, OK, so that is a sort of problem-- it's a limit. But that limit will carry forward to the stuff we do.

So you're going to have to-- right now, in Bitcoin, you can find someone's address, send them the coins-- you're good. With this system, there will be more interactivity between sender and receiver. So that's not great, but not a deal breaker. So we will need to do that. Other problem? Yeah.

AUDIENCE: A key for every amount.

TADGE DRYJA: Yeah.

AUDIENCE: [INAUDIBLE] because of the fact that there can be a lot of different ways to split up the--

TADGE DRYJA: Yeah, it could get bigger. And the actual solution gets big. So that's not great either. But the one that really screws it up. [LAUGHS] Yeah.

AUDIENCE: How do the nodes verify the transaction?

TADGE DRYJA: So the nodes don't verify the transaction at all because they can't see anything. Well, I can do that. Well, how about I just make 70 coins and 2,000 coins? I can make as many coins as I want. No one can see anything because the network just sees nothing-- have no idea how many coins are being sent.

But I'd be like, hey, hey, I'm making more coins all over the place. [LAUGHS] And this doesn't work. Because if I can send coins to myself and multiply them indefinitely, and it's hidden-- so to some extent, you're like well, maybe the network doesn't care. Because well, I'm making them, but they're only my own coins.

But if the network doesn't see anything, it's easy to create the coins. And if those coins are later used, I can't tell if they were made up or not.

So if I'm accepting-- so if I see this, and someone's like, hey, here's 2,000 coins-- I'm like dude, you're making them up. You can't write extra zeros on $100 bill and be like here you go. This is-- come on. No one's going to accept these later.

I won't even know. Did you even have 7? Does this 7 coins come from something where you did the same kind of thing before?

So, basically, if we do that where we completely obscure the amounts from the network and from all third parties, then anytime I'm going to accept this, I'm going to have to enforce the rules myself. And then retroactively enforce the rules for all transactions beforehand to make sure nothing like this happened.

So we could do that. It's all obscured. The third-party-- the network doesn't see anything. And it's up to the participants themselves to say, well, if you're going to send me coins-- since I know if I'm sending it to someone later, I'm going to have to provide some kind of proof that this kind of nonsense didn't happen.

I'm also going to need proof that all of the ancestor transactions that led to these 2 and 7 coins-- I'm going to need those amounts.

And, so, yeah, you could maybe do that. You could provide the decryption keys for all previous amounts. But that basically gets you back to where we are right now-- not quite, but pretty close.

Either you allow people to create coins, or you basically are revealing close to all previous amounts eventually to everyone.

Because as that transaction graph gets bigger and bigger, I'm going to start having all these thousands and thousands of ancestor transactions that I'm going to have to require proof for. I'm going to require all those decryption keys to make sure nothing like this happened.

And then eventually, all the people receiving money will be able to see everything-- well, not quite everything, but a lot.

So it's like well, third-parties-- the miners or maybe people who are just looking at the blockchain who aren't really receiving any funds-- maybe it's hidden from them.

But once you start using the system, you're going to basically be able to see everything. So that's not great.

And it would also be really annoying because then when you have to receive coins, you have to go back through thousands of transactions to make sure that didn't happen.

So we need to prevent coin creation while still keeping the amount secret. This is a tricky problem-- any general ideas here?

So wouldn't it be cool if we could do this where the network sees there's w coins here, there's x coins here, and there's y coins here, there's z coins here? And then we have a proof-- w plus x equals y plus z, without telling you what w, x, y, and z are. Cool. Let's do it. Any ideas? [LAUGHS]

So this seems like this would work. This would do it in that if we can reveal like say, hey, the person d-- z is 2. Cool. So I got 2 coins.

And they can also verify that these amounts are correct. And then if you're-- you could either keep this proof only to the receiver, but then you'd have to give proofs about all the previous transactions, or you can just, say no-- the whole network enforces these groups. This is now a consensus rule where if you want-- even if it's not your transaction, it might be a parent of your transaction someday. So you have to verify it. If you're going to accept the block-- verify these proofs for every transaction. So that would work.

AUDIENCE: So that's essentially a minor deal, right?

TADGE DRYJA: Yeah, a miner, or-- the miners would do it and also the full nodes. If you're running a node-- well, I'm not receiving anything here, but I still need to make sure this is a valid transaction in order to accept it.

AUDIENCE: Isn't that a problem then, because what if a miner is in that transaction?

TADGE DRYJA: If the miner's receive-- the miners D in receiving this coin?

AUDIENCE: Yeah.

TADGE DRYJA: Then they'll know-- then they'll both get the proof, which is sort of redundant for them. And they'll also get the actual amount-- z.

So they'll know this is 2. So that miner would know more, but that's OK. I guess other miners wouldn't-- or other nodes wouldn't. So if you're participating, you'll potentially know more. But the idea is third-parties see a proof and don't see the amounts. Yeah.

AUDIENCE: So SPV will have no way of knowing if a transaction was valid until it was [INAUDIBLE]?

TADGE DRYJA: They already know. So SPV, while it's already have no idea how many coins are coming in on the inputs because they don't have a UTXO set.

And the coin-- so if you see an input, you see it's pointing to some txid. I don't know how many coins there are. I would have to-- you could provide a proof to an SPV wallet that, hey, there were-- I can give you that previous transaction, and you can see how many coins, but then what about the one before that?

So, generally, SPV wallets can't enforce that amounts are correct or verify signatures, because they don't know enough about these inputs, whereas full nodes-- when you tell me to input 0, I know how much. I know what keys-- things like that.

So SPV can't well-- no there's maybe some way to make the SPV have it so that SPV proofs are compatible with this. But it's-- you already don't have that. So I don't think there's much idea-- there's much much research into how to make this SPV compatible. But yeah, it's a good question. Other basic ideas?

So how do we do this? [LAUGHS] So this will take a little-- the actual way to do this will take the rest of the class. Because there's a lot of things where that seems like it'll work, and it's like oh, no, it doesn't.

And this is sort of replaying the ideas from several years of people trying to do this. So how will we do this? So there's an idea called commitments. And the simplest form of a commitment is you commit to some value-- you ouput it-- call it c-- and then you reveal the value and then someone can verify. Well, given c and the value you just revealed, is that a correct commitment?

So I'm committing to the value 2. I'm outputting something c and then revealing it was 2. And let me throw this into my function. 2 and c-- do those match? Oh, it's true-- we're good.

And then there's certain properties you'd want. You don't want people to-- I think I have slides about that. So the hash function is a commitment.

I can say the hash of 5 is 68 empty-- whatever. And then I commit to 68 something. I publish this, and I tell everyone I'm committing to something. I'm not going to tell you what I'm committing to exactly, but here's this 68fde0b7-- some long hash output. And then I reveal days later the thing I was committing to-- that was 5.

And then everyone can check. Let me throw 5 into my hash function. Oh, yeah, it was this. True. So that's a commitment scheme.

It is binding. So there's binding and hiding, and there's different properties of commitment schemes. So this we could say is a computationally binding commitment scheme.

Once I've committed to this, I'm not going to be able to find another number. So I'm not going to be able to say, oh, it was 6 And then get the same commitment out.

So I'm not going to-- so in practice, this is really long. This is a shot 2 output. I didn't want to write the whole thing, but it goes to here, or two lines.

So the idea is I can't find another number, especially exactly the number I want. But I can't find any other number that will get me to that same hash output.

That's a collision, and I'm going to have to try it. Wait, so for a specific number, I'd have to try 2 to the 256. For any collision, I only have to do 2 the 128.

But still, both of those are really big numbers. I can't do it. So it's computationally binding. If I had some amazing computer that could perform an infinite calculation, I could find a collision.

So there's the distinction between computationally and information theoretically. Information theoretically is stronger where even if I had the most powerful computer, I still couldn't do this, whereas binding, in this case, it's computationally binding-- computationally is good enough.

Everything in Bitcoin is already-- everything in all these systems is already basically computational. There are some schemes, which are not computationally bound.

So one of the parts here-- and the Pedersen commitments are not computationally-- and things like severe secret sharing and a couple other things where this is not dependent on processing power.

Even if you have infinite processing power, you just can't figure out what the values are, which is cooler, but it's nice to have. It's not a requirement for our systems.

So this is binding. Any ideas of what this is not-- why this would not be a good commitment scheme? You want to commit to a value and then later reveal it.

You don't want the person who's committing to be able to change their commitment. What else might you want that this probably doesn't provide?

AUDIENCE: Do you need some way to be able to hash the commitment?

TADGE DRYJA: Yep. So you want to be able to do fancy stuff with the commitments. Before we even get there, that's in 5 or 6 slides-- before we even get there, this doesn't satisfy something else we want. Yeah.

AUDIENCE: All but like 5 is such a common number?

TADGE DRYJA: This is easy to guess. So if I'm committing to a number, I'm going to commit to the number of pizzas I'm ordering at my party. And everyone's wondering how many pieces are you going to order? I try things. It's binding, but it's not hiding.

So verifier can easily guess and check the committed value. So for i equals 0. Just keep trying different i's and see if the hash matches 68fd even. And this will take very little time because 5 is going to happen pretty quick.

And then everyone will know he's ordering five pizzas-- sounds like a decent party, but not really worth going out of my way to go to. If it was fffff pizzas-- that would be something else.

So solution-- pretty straightforward solution. Add a blinding factor. So we're going to call that, r I guess, r is random.

So r is b8bc7579. And now I'm going to take the hash of 5 and concatenate r, and now this is a different hash. And now to reveal what I was committing to-- you reveal both 5 and r.

And you need to tell people what order you're doing things in. The system needs to be predefined.

When you're committing, you also have to commit to the algorithms or have pre-agreed upon algorithms, because otherwise, you could do something-- no, I committed to r. 5 was my blinding factor.

You need to have protocols set up beforehand, which you never know, that could be some-- if you say, I'm committing. Here's this value.

How are you committing? What hash functions? What are you doing? But that's obvious. It might not be super obvious, so you have to be careful on that.

So this is now blinding. And it's hidden, and it's still binding. You can't change it to 6. I think that's the next slide, wait.

So you can't change this to 6 and change to a different r. I mean, you could, but you'd still have to do a lot of computation.

And this is now-- some things get easier. So before if you really wanted to reveal that it was 6, you basically can. Because there's a single output for 6, and it just doesn't match this.

So there's not-- if you have a single chosen value that you want to reveal that's not the one that you committed to-- you're basically stopped. There's no way to get around it even computationally.

With this-- that's not true. It's like I really want to reveal 6, I can come up with a different r such that the hash of 6 and our prime is that. And then that's now computational-- slight distinction.

It's still binding computationally. But before, chosen values may not have been. So that's, I guess, the difference.

It's computationally binding for any value. But for a specific value, it might be-- not information theoretic, but you can't do it, because the hash function doesn't help you there.

So what else do we want to do? You're saying we need more. We want to be able to prove things about the commitments. So we need some kind of homomorphic commitments.

And we've seen the homomorphic properties of these elliptic curve points, which are really nice-- where you can add private keys or add scalars, and that's the same as adding these points on a curve.

So let's try that. We want something like this. We want a-- commit to x, commit to y, and get these commitments-- a and b.

And then we want to reveal z, which is x plus y. And then we want the verify function that's put z in and a and b-- either a plus b or a and b separately and return true.

So we don't want to reveal x and y separately. We will reveal the sum of x plus y. And then we will reveal-- and then we'll have the commitments themselves.

So since these are the same operators, it'd be cool if that worked. This is what we want. So that's not that bad. But we also want all those properties where it's hiding, and it's binding and things like that.

So what should we try? First idea is-- this would be useful. How can we build this? We want to-- we can reveal sums without revealing individual parts that we've committed to.

So probably the first idea is let's just use private keys and public keys. Let's use a times g-- that kind of thing.

Oh, oh-- I forgot the joke-- was going to say, gee, how can we build this? g-- it's g. You want a commitment to act. You want a commitment to y. You want to reveal z is x plus y.

So what if you did this? You said, I'm going to make a point on the curve-- x, which is just my value x times g-- point on the curve y, which is my value y times g. This does have that homomorphic property. But is this binding?

It is. I can't come up with a different x that gets me to that point. So multiplying by the generator point is a hash function in that you're not going to be able to find collisions.

So you can't find two different private keys that give you the same public key. Sometimes you can. [LAUGHS] So I don't want-- in this specific case, this is binding. There's a bunch of asterisks that's there. I probably shouldn't. It gets complicated.

There are some curves where it's not. And some systems where it's not. And there have been problems in currencies where people assume that this is the case-- I think in Monero. But, yeah.

AUDIENCE: Can you draw an example of curve?

TADGE DRYJA: Yeah. So they-- the curve for Bitcoin, I think it's y q? No, y squared equals x cubed plus or minus? Minus 7, I think. That's it. It looks sort of like this. It's symmetric, which that isn't.

I think I had it in an earlier one. And then the x-axis is like this is the origin-ish. And then the idea is you have points and you can-- the sum is if you have the sum there it's going to be here, except it's here.

So the idea is 3. Any line you draw more or less will intersect at 3 points unless it intersects at a tangent, or if it's straight up, then you say it intersects at the point at infinity.

But any random line you make will probably intersect in 3 different points. And so then if this is a, b, and c, we define a plus b plus c to be 0.

So then we would define b plus c to be here-- negative a. And the negation is you flip it over the x-axis.

So I talked about this a little bit before. But the idea is you've still got this nice property where you pick a point on here-- g-- everyone picks a random point, and it's a generator. Because if you keep adding. So here's g-- sorry-- and take the tangent-- find this is 2g. Take the tangent here-- maybe it goes here to 3g. Take the tangent here-- or no, that would be 4g-- sorry-- this would be 8g.

So you can keep adding the g, and you'll eventually cover the entire curve and get back to g. So that's why it's sort of a generator because it lets you get to every point by just multiplying.

So the nice property here is if you have some number x times g, and some number y times g-- you can also do x plus y times g, and that equals xg plus yg.

So you can add the points. And that's the same as adding the scalars before you multiply by g. And that's what basically we use for everything.

And then the idea is you can't go back. Given a point, let's say we call this big X-- given a point-- big X, which is little x times g-- you can't figure out what x is. So you can't divide by a point.

You can't say oh, given x, I want to do x divided by g, which is little x. I can't divide some stuff. I can multiply by repeatedly adding. But division is not defined.

So this is binding, but I don't want to-- there's asterisks that's there. But it's not blinded. So same problem is the raw hash-based commitment where if I want to commit to the number 5, and 5 times g, and that's my point that I'm going to use for my commitment.

Same problem-- 4-loop will pretty quickly show you that was 5. How about-- this seems like a good idea. I should say it in the less obviously-- it's not a good idea way. [LAUGHS]

So why don't we add a blinding factor? What would the problem be here? I want to commit to 5. I'm going to add some random number r and then multiply that sum by g, and then when I want to reveal that I was committed to 5-- I reveal 5 and r.

AUDIENCE: You don't know which one is [INAUDIBLE]?

TADGE DRYJA: No, but I can say, look, the first-- well, yeah, the problem is addition is easy to break. When I

committed to 5 plus r, I revealed 5 and r, but it's not binding. I can find r prime, which is 5 plus r minus 6. And then 6 plus that r prime is going to be 6 plus 5 plus r minus 6.

I can compute whatever r prime I want and whatever number I want to reveal later. So this isn't binding anymore.

I'm now adding a random thing in, and I'm committing to the sum of the thing I want and a random thing. But it's not binding because that addition, I can play with.

So how about this? I'll hash 5 and concatenate r and then multiply that hashed output by g. Maybe, but then it's no longer homomorphic-- then I can't add.

So if I have the hash of 5 and r1 times g and then the hash of 7 and r 2 times g-- I can't add these anymore And then have the 5s and 7s add up to 12-- it won't work. Because now it's a hash. And I've just got these hashes adding, and you don't really get any meaning there. So that doesn't work either. So we're stuck.

Then the way you do it is totally non-obvious. And introducing g's fraternal twin-- h. [LAUGHS] You make another point on the curve, and you call it h, which is the letter after g.

It's another generator point distinct from g. So you pick one-- here's h. You also need to make sure that nobody knows n such that n times g equals h.

That n might exist, but you want to be sure that no one knows what it is because you know how to move between g and h, then that'll break this.

But if you take two random points, g was randomly chosen, and you chose a random h, this shouldn't be possible. That would be breaking the discreet log problem.

So I shouldn't be able to compute h divided by g-- you shouldn't be able to compute that.

And so I think in a lot of cases, they'll define h as-- take the hash of g's x-coordinate and then assign that to the x-coordinate for h-- something like that. So it looks pretty random-- should be able to find it.

So we've got another arbitrary point on the curve that we're going to start using for multiplying things.

And so this is basically a Pedersen commitment, which sounds involved and it gets involved,

but really it's not. At its core, it's not too bad. It's saying we're going to have a blinding factor in the actual value we're committing to.

And so now I'm multiplying the blinding factor by g and the actual value by h. And I'm just adding those two up.

So you know how we can do that? If my actual value is 5, I take 5h by taking the tangent-- getting 2h-- take the tangent again-- getting 4h, and then adding 4h and h by drawing the line between them-- I'll get 5h. And then I'll also get r. r's going to be some giant random thing.

R's not going to be like 5. r's going to be a 256-bit random number, and that will obscure what h was-- sorry. what v was because I added them together.

So this is binding unless I know the ratio. Unless I know h over g, or g over h-- whatever-- same thing-- I can't come up with another r and v pair that gets me back to x.

So I can try. There is one. There's probably bazillions of-- if I could find a different r, I could then reveal that v was 8 instead of 7.

But I can't find that r, because I need to know some way to get between g and h-- some factor n where n times g equals h. If I don't know that, I can't do this. It is binding.

It's also hiding. So you might successfully guess-- I might commit vehicles 5-- do some random number r times g plus 5h, and you might guess correctly. But this should have been g-- oops-- sorry. This is g.

But if I have some really long, random number times g, it's also an x, and you're not going to be able to guess that. So this is also hiding.

And it's homomorphic. So x is r1g plus v1h. y is r2g plus v2h. And what if I want to prove that z is value 1 plus value 2 without revealing them individually? Can you see how I could do that?

I can reveal both r's individually. Well, no, sorry-- I cannot reveal both r's individually because then you might be able to figure out the values.

But I can reveal the sum of the r's and the sum of the v's. So I'm going to define a point z, which is point x plus point y. And that's just going to be the sum of the r values-- the sum of those random values times g plus the sum of the actual values I've committed to times h.

And then I reveal the sums. I reveal this-- I reveal that. And then the verifier can do the math themselves. It's a little hard. I'm revealing r and v, which is equal to this. But I'm not revealing these separately. I only reveal the sums.

And then the verifier can check-- is this r they gave times g plus this v they gave times h equal to these two points added together? And if it is, they've proven the sum.

So, for example, it's binding, hiding, and homomorphic. So this would allow us to make proofs.

Binding, hiding, homomorphic-- you reveal the sum of the r's. You reveal the sum of the v's-- it works.

So basically now instead of coin amounts, which are 8-byte-- they're in 64s-- I just provide elliptic curve points, which are 33 bytes, so it's a bit bigger-- not a huge deal.

I can put another 20-something bytes in my outputs-- no big deal. And then I provide a proof by revealing these sums each time.

So I prove that w plus x equals y plus z. And everyone can see this little w amount of coins-- there's actually y coins. There's actually x coins. There's actually z coins. And there's a blinding factor that all the participants have to keep track of.

So the receiver's views-- they learn their own v and r. When someone sends, they would have to tell the receiver, hey, here's r4. The sender makes up r4.

And they can just compute-- let's make a random r3. And then compute r4 such that r4 plus r3 equals r1 plus r2.

They need to make sure that the randomnesses add up, which is easy because they're regular scalars, and the sender has full control over r3 and r4.

So, basically, make random ones until you get to the last one. And then make the last one such that they all add up to the right number.

And then the sender can say, here's your r4 and here's 2. You're getting 2 coins. And the receiver gets those two numbers and says, I can see that this transaction commits to z-- a point.

And you gave me an r and a v that when I do the operation-- I get z. So I know that

successfully, you've revealed that these are two coins. And I can use these values later when I want to send my own coins.

So this works, but there's more problems. [LAUGHS] Any idea what the next big problem is? Yeah.

AUDIENCE: If there are only two outputs, and then c's r4 and d4-- they could be able to [INAUDIBLE] r3 and d3, right?

TADGE DRYJA: Yeah. But that's OK. I mean, there's no way around that. If there's only one input-- or wait-- if there's only two outputs, so if the receiver learns these, they will be able to distinguish-- if you know the total amount of inputs, and there's only one other. You can just subtract and figure it out.

So there's all sorts of things. If there's only one output, that person knows it's the sum of all the inputs.

AUDIENCE: You don't know-- in this case, you don't know the total input, right?

TADGE DRYJA: Right. In this case, they don't have to reveal to you the v's-- they just have to reveal that 2 plus y equals w plus x.

So you don't have to learn their input amounts. If there's only one output, then you learn if this is 2, these must have added up to 2. But I don't learn exactly which.

So this is useful. There's still a big problem. r1 plus r2 is r3 plus r4-- you can prove w plus x equals y plus c-- it works. It's great.

AUDIENCE: Do nodes only mean the bigger transaction-- to know that the inputs you put [INAUDIBLE]?

TADGE DRYJA: Yep. nodes can just compute these numbers, and they're good. So one thing you could do here-- OK, wait.

So there's one thing that you can do to break it. But it's not a big deal. And then there's another thing-- you need a break that's a really big deal. Yeah.

AUDIENCE: Since you're proving sums, is it that you can have a higher likelihood of proving something that doesn't-- like that wasn't actually true?

TADGE DRYJA: Yeah, but how exactly? So one thing you could do is-- I think, I said-- so, basically, you can

verify it and put in outputs-- add up all the points-- make sure they're equal.

You reveal r and v to the person receiving. And you don't want to forget r. r is now a private key. If you forget r-- you're stuck. You can't prove anything anymore.

You can make invalid outputs, which are just points with no known r and v. But nobody should accept those. So you can do this where-- I have no idea what-- there is no commitment here.

Maybe these have valid commitments, but this is a point, which is w plus x minus y. And there's no known r values or v values for these things.

And from the network's point of view, this will still be valid. The networks are saying, well, w plus x equals y plus z-- we're good.

But then anyone accepting that transaction-- you're not going to be able to give them an r value. And so they're not going to say this is actual money.

So this is possible, but it's not a huge problem because you can destroy your own money. You can already do that in a million different ways in Bitcoin.

But if you send to it to an output amount that doesn't have an actual r and v value-- that doesn't have a valid Pedersen commitment-- those coins can essentially be destroyed because no one's going to be able to figure out r's and stuff for here.

So that's one issue-- not a big issue. Just don't do that. Don't ruin your own money. Make sure you always have valid commitments.

And when you're accepting payments always make sure that you're given the values and the random blinding factors. So that's one little risk but not the big problem.

Anyone have an idea about the big problem? So as a hint-- all of these things, and I haven't written it because it gets really redundant-- all of these things are like mod p, which is some giant prime number that's 2 to 56 minus 17, or it's not-- in the curve, ed25509 it's 2 to the 255 minus 19. So that's an easy one to remember.

So that's why they call it curve 22519. In bitcoin, it's 2 to the 256 minus some numbers that are not too huge.

So since all of these things are modulo-- some big prime number-- what's a potential problem

here? It's a big problem or in some ways the opposite of a big problem. Not a small problem. [LAUGHS] The opposite of a big problem but not small-- big negative problem. [LAUGHS]

You can basically commit to negative values. So I send negative 99 coins here, positive 108 coins here-- the numbers all work out. So you don't have negative.

Usually, if you're doing mod-some integer, you don't think of it as negative, but you can think of being almost right at the top of that modulo range as being equivalent to being a negative 1.

So if you say, I'm going to have p minus 1-- that's some huge number that's within the very allowable range. But it also will let you treat it as a negative 1 for operations like addition and multiplication.

So this would work, and you can calculate what negative 99 is-- that's just going to be p minus 99, or minus 100.

I always am off by 1 when I do these things. [LAUGHS] Wait, is it 100? Anyway, so you can calculate-- it's not really minus 99-- it's some big number that's going to be 32-bytes long-- that's modulo.

Modulo p doesn't change, but it's equivalent to minus 99. So that if I do add these to this huge number plus this other small number, it loops over that modulo and then gets back to-- what, 9.

So this plus this is going to be 9 even though this is some really long number. This plus-- this is going to be nine, and then it all adds up.

And then I've got this either negative or implausibly large value that I've committed to, but I can just ignore that. I can get rid of that.

That negative output will be hidden. No one will see that I've got this negative 99 here-- then I can toss it and end up with a whole bunch of negative values that are anti-coins and a whole bunch of more new coins.

And later on, I can prove I've got valid proofs-- all these w plus x equals y plus z proofs worked the whole time. Shoot-- [LAUGHS] so we're stuck.

We need more than a proof that the sums are equal-- was working really well. But just a proof that the sums are equal is not enough because we're modulo-- this big prime number. And so

you can make these negations.

We also need a proof that they're non-negative, and not just not negative, because that negative 99 is essentially p minus 99, which is some huge-- it's 2 to the 256 minus 99-ish. So it's not just that they're non-negative-- we have to prove that they're reasonably sized-- that they're fairly small.

So if you could prove this number is non-negative, and it's less than 2 to the 64-- that would be cool. How can we prove something about the number itself without revealing it?

So now this is an even trickier problem, because, before, we wanted to prove a sum. We wanted to prove that z equals x plus y.

I don't want to tell you what x plus y is, but I'll tell you what z is. That was proving a relation between these numbers. This is just proving something about that number itself individually.

We have to prove individual numbers-- ranges. So we have to say, this to 2h-- there's a 2h in there, and I can't show you it, because I'm just committing a w.

But I want to somehow show you that the v here-- which should happen to be 2, although I'm not going to tell you-- the v there is within a small range-- without telling you what it is-- without revealing it-- tricky.

So what you can try to do is sign with the points. Somehow, this is what'll get us there. The idea of we've got these Pedersen commitments-- we've got these points on a curve-- those are public keys. That's how we used to usually think of them.

Can we sign? We know the private keys. We know r1. We know 2. Can we make signatures somehow and somehow use that to reveal some information about h but not too much?

So the signature equation-- I didn't really review as much, but we've talked about this a couple times. Come up with a random value k-- take the hash of k times g and the message to be signed times your private key-- that's your signature s.

The problem is x. Your value that you've got committed to here-- it's r2g plus 7h. What's your private key here? Well, it's our 2 plus 7. But, no, because you know the private scaler's here. But there's 2 points involved in this key.

So you can't make a signature. Generally, signatures are with respect to a single generator

point.

The way to verify this is to multiply both sides by g. s times g should be k times g minus the hash of k times g, message times 8 times g which is a public key-- that works.

But now you've got h in here. So we're stuck. We could try to define another signature thing with both h and g, but that doesn't work.

But what we can show, which is kind of interesting, well, what if v is 0? Then x is r2 times g plus 0h, which is just r2 times g.

This is nothing. There is no h here. And that means my private key's going to be r2. Well, now I can sign-- now I can re-sign with respect to g with this point x that I've committed to because the value is zero. The value-- the thing that I'm multiplying by h is 0.

So it'll work for a signature system. The same regular old signature equation that we use, I can sign a message with key "big X." And so this is a proof of a 0 value-- I'll sign with my own key.

So if x is some random blinding factor times g plus 0 times h-- where v, the value you're committing to is 0-- well, that means that the signature is just-- so, the question is, what message do you sign? Well, maybe sign the message which is the pubkey itself. You don't need to sign a message but throw that in there.

And then this will work. And someone can say, hey, here's a signature from key x. And the signature is arbitrary. It can be a signature of any message you want. But the fact that they are able to produce a signature reveals to you that v is 0.

So that's cool. It's not that cool, right? Because, well, I just revealed my value, and it revealed to be 0. But I did it without revealing r. So that is cool, right?

I could do it by saying, hey, here's my r and v values-- multiply them and add them up to see that it gets to the point I committed to. But I can now prove that-- I can prove something about v without revealing anything about r.

Before-- we were saying, the way you reveal these commitments is you reveal the blinding factor, and you reveal the value itself. Well, now I can keep the blinding factor secret and reveal a little bit of information about v.

This works, and you can't sign if h is non-zero. You're not going to be able to-- you're not

going to be able to calculate any valid signature unless v is-- ah. V is non-zero. Sorry.

So you can also do this with a proof of v equals 1. Again, sign your own key. So you define x prime to be just x minus h.

And then you can say, well, look, I'm going to prove to you that v is 1, and the way I do that is take the point I've committed to x, subtract h from it, and I'll

provide a signature on that new point. Well, this should be v. I can't do that unless v is 1. So if I have this, and then I subtract h, then I get back to r times g. I can now provide a signature just using r-- it'll work.

So what does this get us? We can prove that v is 0, or we can prove that v is 1, or, really, we can prove that v is anything. Just subtract a whole bunch of h's and subtract 7,000 h and now provide a signature. And that proves that v was 7,000.

But wait, we just revealed v, so what's the point? Did that really get us anywhere? If we can make a signature that reveals the value we've committed to with hiding the rest of the stuff-- is that useful? If it's kind of cool, but what can we do with this? Any ideas?

So there's another part that I didn't have time to introduce, but maybe will at some point, and you can read about it. It's really cool.

Ring signatures-- a ring signature is similar to a normal signature. You've got key pairs. You've got sign, verify, but there's a set of public keys during the signing and verification process.

So the basic idea is, I sign a message with one of the public keys, but I don't tell you which one. And you verify that, yep, there was a signature here. I'm not quite sure who signed it.

There's possibly three or four different people who may have signed it. But I know one of them signed it but not which.

So, basically, it's the key-generation step-- same as before. You have some keygen-- takes in some randomness, and it spits out a private key and a public key for you. And you can use those later.

Sign, on the other hand-- sign function used to have the message to sign in your private key, and it would output a signature.

In ring signatures, you sign with the message you want to sign-- your private key and then pick a list of other people's public keys.

And this list can be just a single one. It can be 1,000. You pick a bunch of other people's public keys, and you generate a signature.

And then verify, you use the signature to-- take the signature in-- the message that's been signed, and that list of pubkeys.

And it outputs a Boolean of, yep, that signature worked, or, no, it didn't but it doesn't tell you which of the public keys is signed.

So this is a really cool idea. I think the initial paper was How to Leak a Secret, which was-- and then there's different papers about different more efficient ring signatures.

But this is a really cool thing to use. I think this could be really cool for a lot of different things. If people had public keys that they were associated with and wrote them on their business cards and stuff like I do-- you could then have leaking information that's more credible.

So instead of just the newspaper saying, sources close to the matter said x, y, z, or a top official at the White House said this.

You could have it being signed, and we've got all these different White House officials' public keys-- one of them signed this message.

So either there was a key compromise, and they all lost their keys, which is, in practice, probably more likely in that case, or one of these people signed this message, and I know it's one of them, but it doesn't reveal anything about which of the public keys-- so, that's pretty cool.

So you can verify it's from a key in the array of public keys but not which. So let's put these together. I can sign with x to prove that v is 0. I could also sign with x prime, which I'm defining as x minus h to prove that v is 1.

Well, what if I have a ring signature where I say I'm going to produce a signature from both of these-- from one of these 2 public keys. And that would prove that v is either 0 or 1. But it doesn't prove which. Make sense?

So the idea is, I can make these signatures that essentially reveal what v is. They indirectly

reveal what v is by saying there's no way I could have made this signature unless v was 0, or unless v was 1 because you're-- you know that the h component-- v is the coefficient for h.

So you need to know that I don't have an h component in there if I'm going to make a signature with respect to g.

And I can make any number of different x primes by subtracting different amounts of h. But a single signature would just reveal what v was, which isn't useful.

But a ring signature, whereas there's 2 different pubkeys-- one of them's x-- one of them's x prime, we just subtract h. And I'm going to provide a ring signature from one of these 2 pubkeys.

Now I've proven to you-- well, v is either 0 or 1-- it can't be both. But it can't be 2. It can't be 3. It's got to be one of those two values. So does this make sense? Questions on this? Yes.

AUDIENCE: What does unverified look like? How does it verify that?

TADGE DRYJA: Oh, well, the original scheme is kind of complicated. The one used in confidential transactions is Borromean ring signatures.

You do a bunch of hash functions. I don't know it. I don't even really know it well enough myself to really be able to explain it in a minute.

But, basically, you compute something on all of the pubkeys, and then verify the signature on the aggregate pubkey as a way to think about it. The signatures can-- the computing time gets big with a number of pubkeys.

So if you have millions of pubkeys that could possibly be signing-- the verification algorithm gets really time-consuming. So I can maybe go into ring signatures another time. Yeah.

AUDIENCE: I thought the point of this was to prove that v is non-negative. So right now, we're just saying it's not-- it could be two values. How does it help us through the signs?

TADGE DRYJA: Well, neither of these are any good. So we've just proven-- we've done quite a bit more than prove it's non-negative. We've proved that it's exactly 0 or exactly 1. We might want more values.

So make a ring signature from a million different public keys where the pub n is just pub n

minus 1 minus h. And then I just proved that v is somewhere in the range from 0 to 99999. Yeah.

AUDIENCE: What is computation time for ring signatures as the number of public keys?

TADGE DRYJA: Yeah, it goes up. The best is linear, or, no, there might be-- sorry-- there's probably sublinear now. But this is not feasible. [LAUGHS]

The traditional ring signatures are linear with the number of public keys to be verified. So if you did this-- well, all even then, you'd have to throw, though, because-- you have to at least throw the public key into the verification function. And so that means you need to compute a million pubkeys-- throw it into that function. It's going to take a long time. So this is not itself going to work. Yeah.

AUDIENCE: Just Monero uses ring signatures?

TADGE DRYJA: Yes, Monero uses ring signatures for a different reason. Well, now they use it for this kind of thing, too.

The Monero ring signature was initially used for not hiding the amounts but hiding which outputs were being spent.

So in Bitcoin, everyone should-- So in Bitcoin if you have a transaction-- here's my input, here's my output. And the simplest one is just-- this points to tx5 or something, and then my output is address-- and it's address b, and it's got 7 coins.

Everyone can see what you're spending. The idea in Monero is, I don't point to an input-- I just provide a signature. And it's up to you to figure out where that signature's from.

So I provide a signature and a list of pubkeys-- pub a-- pub b. So the smallest would be 2. And then I provide a key-- key in a number.

So the idea is instead of thinking of it as outputs, you think pubkeys have associated values. You don't look at the transactions themselves.

This is key c. And there previously have been-- pubkey a and pubkey b have had values-- 7 coins sent to them.

And I can now say, look, I'm going to provide a ring signature from either pubkey a or pubkey

b. And I'm not going to tell you which, but you can see that both pubkey a and pubkey b have received 7 coins in the past. I want to send 7 coins over here. I'm going to prove that I own one of those two. And then you can validate that the transaction's valid.

And then if later, you see another transaction with a ring signature from pub a-- pub b-- also sending to the key d-- 7 coins.

Now, I can delete pub a and pub b. I don't know which, this-- so in transaction 1, I don't know whether a or b was spent, but one of those two was. And then in transaction 2, I don't know whether a or b are spent-- one of them was.

But anyway, they've both been spent now. And so I can delete them from my storage. But the ring signature's in Monero were used to obscure the transaction graph itself-- where you can see that there's a bunch of different keys that have money associated with them. And I'm going to provide a signature that I own one of them, but I don't tell you which.

And now they also do confidential transactions to obscure the outputs using ring signatures as well.

So I don't-- well, it's interesting. I haven't looked at it that closely. So I know that's the basic idea. But there's a lot of subtle details involved in how that works, and I'm not super familiar with that.

So I can finish up. So the way to make this practical is, let's say you have a ring signature for each bit, so you're going to have to have a different signature for each bit in your value.

So the pubkey x0-- well, I'm going to have a ring signature where I prove that it's either 0 or 1. And here I'm going to prove it's either a 0 or 2. Here I'm going to prove it's either 0 or 4.

And so if I have 32 signatures, I can now prove a 32-bit number. Each of those individual signatures only has 2 public keys associated with it. So that's now more feasible.

There's a total of 32 public keys, 32 signatures-- not the end of the world-- still kind of ugly. And then there's optimizations where since these pubkeys are all the same, the 0 value for that bit-- we can squish them together and stuff to make it a little bit smaller.

So you have a signature per bit of your output. If your values are not too big-- this works. We're going to limit it to 32-bits. So from 0 to 4 billion.

Now you can prove that each bit was either a 0 or 1, essentially, without revealing whether it was a 0 or 1.

And then it's a couple of kilobytes per output. So it's a little bit annoying. It used to be 8 bytes. You said, hey, I've got 37 points. Also, these 8 bytes are very sparse in that they rarely exceed 4 bytes.

So you can compact it down on disk. So this is a little-- scalability-wise, it's an issue. You're now going from what used to be 8 bytes is now a couple kilobytes.

I think they've got it down to 2 kilobytes. But still, that's 2 kilobytes. And it takes a bit more CPU time to verify. Because now, instead of verifying one signature to see that the coins are moving to the right place, you're verifying 30-plus signatures for every input and output.

Also, the real problem is it's not really compatible with Bitcoin-- this would be a really tricky fork to implement.

How do you implement this such that the old software on the network is OK with these transactions where they're expecting a transaction that has a value?

Here's how many coins are being moved. If you say, no, I'm not going to tell you anymore. Well, old software doesn't know how to deal with this and by default, will reject these transactions.

So there's ways to do it that doesn't break compatibility, but they're pretty ugly. One of the ways is to say, OK, from now on, all these transactions-- all output values have to be 0.

So the old software will still work, but it will be pretty confused and say, I see all these values. I see thousands of transactions. No one seems to be sending anyone any money. So they seem to be pointless.

And then there's this hidden value that's in a different data structure that would look something like segwit. And segwit was a little bit ugly-- this would be real ugly. Those are the trade-offs we're dealing with here-- tricky fork.

But you get this. This is what you get when you use confidential transactions. You got it, right? We have private, unlinkable amounts where the network can verify-- OK, let me verify that w plus x equals y plus z-- that's the easy part.

Now, let me verify for every individual w, x, y, and z that it is within the range of 0 to 6223222232-- 02 to the 32-- it's a tongue-twister-- by verifying all these separate ring signatures for each bit of the amount.

But you can do it, right? You can verify-- input amounts equal output amounts. All these input amounts and output amounts are normal looking numbers that aren't too big that aren't right up near the modulos to wrap around. And it works-- it's just big and slow-- we're trying to make that faster.

So probably not going to go-- if you're interested in this, though, there's a lot of research about this kind of thing right now. It's really cool getting in the crazy, interesting math.

So there's bulletproofs, which was Benedict's at Stanford. He wrote up-- I mean, a bunch of people wrote it, but I think it was sort of his idea, or his thing working with other people, too.

That's more efficient range proofs. I do not know how bulletproofs work. I tried to read the paper and got two pages in, and I don't know. [LAUGHS] But I'm sure I could figure it out-- it would just take me a long time.

And then the Borromean ring signatures-- the more efficient ring signatures that can compact a lot of data so that you're not-- so you get it down to two kilobytes for these range proofs in confidential transactions.

And then I may try to give a class about this in a week or two-- MimbleWimble. The idea is that when all of the transactions in the network are like this and that have confidential inputs and outputs amounts, the transactions can be canceled out in an interesting way.

Because the output amounts are unique. They have these blinding factors. They have these range proofs. And so if you're in a block-- so just a hint about MimbleWimble-- the idea is if you have a transaction and amounts are all obscured. But a is being consumed, b is being consumed, c is being created, d is being created. These are amounts.

And then I see later in the block there is another transaction which spends d coins and sends to e coins.

I don't have to verify that a plus b equal c plus d. And d equals e kind of thing-- I can just cross these out.

And I can verify on a block-level instead of a transaction-level that the input amounts and output amounts all matched up because I don't know what d is.

But, anyway, it was on an output side and on an input side within the same block. So, anyway, it's gone. It's being added to and subtracted in this block.

So I can do that. And I can do that over multiple blocks. And I can basically keep a tally of how many coins are in the system and cancel things out.

So that lets you do some really cool things where you can skip a lot of verification at no loss of security.

So that's the idea of a MimbleWimble, which is really a fun paper because it was written by Lord Voldemort.

Someone posted a Pastebin link on IRC. And who knows who wrote this-- Lord Voldemort. Except it was Lord Voldemort-- the French word for Lord Voldemort.

So when they translated Harry Potter into French, they changed the names around, I guess, and it was the French equivalent of Lord Voldemort.

And so now there's a bunch of people programming MimbleWimble and implementing it. And they all use Harry Potter-- not all of them, but a lot of them use Harry Potter names on GitHub. So it's kind of funny.

It's also really funny that it was one of the most-- I would say, the most-- whoa, kind of paper of 2016 now.

And so it's of an interesting indicator of the space we're working in that you've got IBM and all these big companies, and they're working on blockchain research, and, honestly, not a ton of awesome stuff, whereas like, whoa, MimbleWimble, OK, Lord Voldemort-- he really showed everyone. [LAUGHS] So it's fun that it's still a very out-there hackery system.

So that's-- hopefully, most people got most of the idea. It's a little bit fast. It sort of explained all the Pedersen commitments and range proofs and everything in a little bit of time.

But I think you got the idea. And if you're interested in it, there's a lot of current research on this and people implementing it. And there's a lot of cool things you can do with it, so.