PROFESSOR: Sorry. I have a bit of a sore throat, hoarse voice. I was talking a lot this weekend. OK. Also, today we're going to do transaction malleability, segregated witness, and I'm endorsing an ICO for the first time ever publicly, and it's Anne's intermittent cookie offering. So if you guys want cookies. It's an airdrop and you just get them. so it's my first ICO I'm endorsing.

OK. So malleability. So malleability is the ability to deform under pressure formateer. And so bitcoin is modeled off of gold, which is the most malleable metal. You can make gold leaf and stuff. So clearly we need to design bitcoin to be malleable. No, I'm joking.

OK. Actually in the context of cryptography, it's not super hard definition, but it started with Cipher text, where if you can modify a Cipher text and that modification will carry over into the plain text when it's encrypted. It also applies to sort of messages and signatures. In our case, signatures can be malleable, which means you can change the signature and it's still a valid signature. So given a signature S1 on message M1, you modify the signature to S2 or S prime and it still signs the same message. It still validates as true.

So when we were defining signatures this wasn't really an attack we'd considered. There's still a signature and you can't forge a signature, but you can dot an I or something and the signature is slightly different. You can't create one yourself, but given a valid signature you can make a slightly different valid signature.

And that's how it works in the bitcoin signing algorithm. But there's all sorts of different contexts where malleability exists in cryptography. And then part of it is things still have to work for whatever definition. So if you malleates a signature and it no longer validates, well, that's sort of a trivial, like-- yeah, sure, you can do that to any bunch of bytes. You can just flip some of them and the whole thing doesn't work anymore. That's easy. But properties where it still works.

So this leads to some weird stuff in bitcoin where you can change a transaction and it's still valid. And that's generally not what you want. If you've got some kind of contract or some kind

of payment and you write a check, you don't want someone to be able to modify the check and still be able to cash it.

And I don't really use checks much, but they draw the line. Like $100 and then they draw a line so someone doesn't write and $0.99 or something after. Not that that's like the greatest attack ever. Or $100 and then someone puts like $1,000. I don't know. But it's sort of like that where someone can change the thing you sign, can change the thing you are agreeing to after the fact, and it's still valid and it does something you don't expect.

OK. So a review of the transaction format, which should be probably in people's minds if you were looking at the homework, the problem set. And one thing to focus on is that the outputs don't look like the inputs. These are fundamentally different things. The outputs specify a transaction ID and the inputs, and then the outputs specify a script and an amount.

There's another 4 byte field here that doesn't matter. So basically you're spending from a transaction and a sort of row and you're spending too just this arbitrary pub key, but you're not spending from a pub key and you're not spending to a transaction. You don't identify your transaction itself here. Almost every website that shows blockchains is it gets this wrong.

So if you look at like, I don't know, blockchain.info is probably the most popular and you just look at a transaction. They don't have it anymore. OK. So you look at a block and then you look at a transaction in the block. We're going. No. No. No, not yet. There we go OK. So you look at a transaction. Yeah, it shows. This address is sending to these two addresses. And blockchain.info is particularly egregious and that there may actually be more than one input and two outputs in this transaction. They hide change transactions.

So it looks like, hey, this address had some money in it and it sent it to these two addresses. And if I click, oh, where did this money come? It come from 18eecz, and it shows here's the bitcoin address, and, oh, it's got multiple transactions this addresses has been involved in. This is not how bitcoin works. They are running their own database and sort of making up this view of the network, which is incorrect. Transactions don't send from an address, they send from another transactions previous output.

And this is very confusing because in the case, let's say in this transaction, there is a-- what is it? 767 something. So it says, yeah, it's coming from 18ee whatever, and if I click on it I get three different transactions. There is a specific transaction that 18ee was involved in that is

being spent from in this transaction. I can look it up because I have an actual full node.

So if I say get raw transaction and I put it in here and I can see, OK, it was spending from c838. It was spending from this transaction, not just an address. So I mean if you're coming at this sort of new it's like, OK, fine, why do you keep talking about this? But if you've been working on these things, a lot of people, myself included, for like six months a year I looked at these websites and I'm like, oh, this is how it works and then six months in or something looking for code, and I'm like, wait, huh? This code is wrong, but no, this is the bitcoin code that actually is running. And so it's a weird thing to sort of get used to. Like no, you're not spending from an address. You don't show the address at all when you spend from it, you spend from a specific output.

OK. So that leads to some weird issues. Specifically, what gets signed? So to some extent you're signing the whole transaction. You sign. You want to sign everything. When you're saying I'm sending money from here, I'm sending it to here, you want to make sure that your signature covers the entire transaction so that people can't add stuff after or remove stuff.

So you want to sign the inputs and outputs. But the inputs contain signatures and you can't sign the signature. That doesn't make any sense. The signature is the thing you're putting on at the end. So it's sort of weird. You've got this document and you have a little line at the bottom for the signature. But should the signatures be maybe a separate page that refers to the previous page? It's actually kind of a weird confusing problem.

So in practice, in bitcoin, what Satoshi did in 2009, you take the whole transaction, but you remove the signature fields. You basically zero them out. Just put a zero there and then you sign that sort of signatureless transaction. And then you put that signature in after the fact. And so that means if you change any bit of the stuff that gets signed other than the signature, the signature will break. So does that make sense?

You have these empty lines kind of, and the idea is you empty them out, you make them blank lines, and then you take that whole message, hash it, including the empty parts, and then paste in those signatures after you've signed. You don't empty out every line, the line that you're specifically putting the signature in, you actually put a different few bytes in there, which leads to other problems that I can maybe mention if I've done.

So this seems OK. It's like well, look, I can't sign the signatures, sure, but if you change any bit of the stuff I'm signing, the signatures now break. So this seems perfectly safe. No one can

change the amounts I'm sending. No one can change where I'm sending it to. No one can change where I'm sending from. All these things get covered in my signature, so I'm good.

Problem. The transaction ID, the way you refer to transactions is the hash of the whole thing. The txid does not zero out the signature fields. So the identity of the transaction itself, the way to point to and indicate where you're spending from, that includes the signatures. So that also seems like, well, that's OK. When I point to something I'm indicating the whole transaction, the whole signed transaction. But the problem is that can change. The signature itself may be malleable, and in bitcoin it is.

There's third party malleability problems. So the simplest one was leading zeros where all these things are numbers. You could say, OK, someone's got a signature. It's this big, long string of bytes. I'm just going to put zeros in the front. I'm going to put a zero byte in the front of it, and that doesn't change the meaning. If someone says, I'm sending you $1,000 and I put a 0 front of the one and 1,000. Well, it's still 1,000. However, for the purpose of a hash function, if you have a zero byte in front, that changes the whole hash.

And so they got rid of this pretty early. They sort of made it so that you had to have the exact right number. You can't have any leading zeros. But the first one was just, oh, I put a 0 in the front. The harder one is called low s, where part of the ecdsa signature scheme. I showed before that it's this curve that's symmetric about the x-axis. Whether the thing you're indicating is on top of the curve or it's sort of reflection on the bottom, it's valid either way. So for any given signature, there's another signature that will be valid. You just sort of flip it, make it negative or positive.

So now there's a standard, OK, you need to have high s. Low s signatures should be invalid. Both of these are really tricky because they're third party malleability. Anyone can just listen on the network, see a transaction, change the signature. And in doing so they will change the txid, which is how all the software refers to transactions. So it looks like a new transaction to most of the software. And the transactions are still valid. The signatures are still valid and you're not sure which one will get in.

There's also first party malleability, or in some cases second party if you're doing transactions with multiple people signing. So I'm not going to go back into the elliptic curve signature algorithms, but there is a nonce. There's randomness that you inject into the signing process. It's not deterministic. It's not that given a message and my private key. I always compute the

same signature. No, that's not how it works.

There are signature schemes like that, but in the case of the elliptic curve stuff that bitcoin uses, you have to make up a random number to make each signature, and no one knows what that random number is. So you could make up different random numbers. You can make as many signatures as you want. So given a message and your private key, you can make arbitrary number of signatures that are all different signatures, but they're all valid signatures.

There is a sort of standard for how to make them the right way not randomly. It's basically take the hash of your private key and the message being signed. Put them together, hash that, and use that as your random number because then the idea is well, it's got secret information in it as well as the message specific information in it. So no one's going to be able to guess what it is so, and it's still kind of message dependent so it'll change each time.

So there is that, but that's something you can do. It's a really good idea because if you use a non-random k, if someone can guess k or if your random number generator's broken, they can steal all your money. They can figure out your private key. So you don't want to be dependent on generating randomness. A nice way to model it is, OK, have some initial event where you're putting in random numbers and you're storing them and then from then on you want everything to be deterministic, then you don't have to rely on random number generators.

So it's really a good idea to use this. And I use it in my software. Most things use this kind of standard. However, you can't verify that anyone's actually using it. It's purely internal. It's a internal way for you to make your own signatures, but no one can actually-- can you prove? No. I'm not going to say you can't prove you're using it, but not in any reasonable fashion. Yeah. So no one knows if you're doing it.

So this is an attack where you can say, OK, I'm going to make a whole bunch of different signatures. They're all going to be valid, but that will mean I've got a whole bunch of different transactions that are doing the same thing. So maybe the question is, what does this really do? Does this really hurt? If someone dots an eye on your check, it's the same amount. It's going to the same place. Who cares. Outputs are the same. Inputs it's pointing to are the same. It's just this sort of annoying thing. OK, I tweaked it and I changed the txid. No big deal.

Well, in some ways, yeah, it's no big deal, but a lot of wallets didn't deal with this well. So let's say you're running a wallet, you make a transaction and you sign and you broadcast

transaction 2d5cac and it never gets confirmed, and instead someone out there flips a bit, changes your transaction to 9cba3e and that gets confirmed, and your wallet just says, yeah, this transaction you sent never got confirmed. There's wallets that did that. Most of them have fixed it by now.

But if you're thinking of transaction IDs as the identity txid, this is the name of the transaction. I create it, I'm watching it to see when it gets confirmed, and I'm not looking for some malleated version. I'm just watching this thing that I created, never gets in a block. Weird, and it's just stuck in the wallet. So there are definitely wallets, and I think everyone's fixed it by now. But a couple years ago, definitely wallets that would have these problems.

It's a wallet problem. Your money got to the right place. You just need to sort of either delete stuff in your wallet or upgrade the software or tell it to somehow forget about this transaction and actually look on the blockchain for everything similar to your transaction. But it did do some damage.

So, I don't know, 2014 the Mt. Gox thing where Mt. Gox got hacked supposedly and lost all the money, they blamed transaction malleability, which was kind of interesting. There may have been an attack on Mt Gox that used transaction malleability. The attack probably was this, which was log into Mt. Gox, withdraw some coins, modify the txid to this, and then it gets confirmed, you get your coins, and then log into Mt. Gox and say, hey, this never happened. My withdrawal didn't work and then their system would automatically issue a new withdrawal transaction.

And so you could just start taking all the money out and your balance on the system's like, well, we keep trying to send you money and it keeps never getting confirmed. And so we keep making new ones. I don't know to what extent that that actually happened. It couldn't have been the whole thing for Mt. Gox definitely. There's still a lot of uncertainty about that.

But it's indicative. If you write your own software and it's not accounting for these things, you may be losing money once people say, hey, this didn't work, make a new one, and then you keep doing that and losing a ton of money. But that's pretty sloppy practice.

Another issue. If you're spending from a unconfirmed change output or an unconfirmed output-- so you make a transaction, you send the two different outputs, you've got a txid. And you're sending five coins to this person and three coins back to yourself. That three coins back to yourself output, you might want to use it again pretty quickly. Sometimes this happens.

And so you've got a change output that's from transaction 1, 7feec1. So you're going to now spend that change output, however the txid of transaction one changes. So you're saying where you're spending from is no longer valid. And this is a big problem because now you've signed a transaction that you think is going to be valid, but the money you thought you were spending sort of moves out from under you. And so that transaction's no longer valid.

tx2 is now invalid. It refers to something which can never be confirmed because there's a different transaction that's almost the same, but they're mutually exclusive that did get confirmed.

OK. So this is bad. It doesn't seem that bad. And so for years in bitcoin this was a problem that while it dealt with, software and people would be like, oh yeah, you have to backup your keys, delete your whole database, and rethink the blockchain and then it'll find the right transactions. Kind of hacky work arounds like that where it didn't happen too much. It wasn't a great attack. You can annoy people. You don't get any money. So wasn't a huge deal, but it was annoying.

But the idea is you can always re-sign. You've got your private keys. If the money you are receiving sort of shifts around and changes its location, well it's still yours. You just need to re-sign. But what if you can't re-sign?

So the case of multisig where in most cases when you're doing transactions if you just have one key, it's just you, that's fine. In the case of multisig usually you're all friends to some extent and you're all in the same organization or multiple devices that you own. But you can have sort of adversarial multisig where you're assigning transactions with people who are you're sort of cooperating with them, but you may not really trust them, or they might be potentially attackers, things like that.

You can definitely sort of extend your multisig model into that. And there could be multisig pre-signed transactions where, OK, we've got this two of three output address, this output that exists, and one of the two or three has pre-signed a transaction and hands it over to me and then they disappear. And they say, oh OK, well I'm going to now sign my side. But if malleability occurs and the transaction ID changes, that signature is no longer valid, signing something that's not there anymore.

So this is very important in payment channels lightning network stuff that I'll get to in a few days. And so it wasn't so much that malleability was like a showstopper bug and everyone was

losing tons of money, it was that it was preventing kind of new, cool things from happening because there were a lot of problems with, OK, let's make this construction where we put money into a multisig account and then I sign like a refund transaction that's got a lock time before I actually fund it and things like that where you couldn't reliably do it because if either party modified their signature, they could break the whole thing and they could have a tax where it's like, OK, we're doing something together. Oh look, it's got stuck.

Well both of our coins got stuck in this place. Hmm. Now it's sort of a hostage situation and you can say, well, I think I should get 1 and 1/2 and you should get 1.5. It's like wait, we both wanted 1. So there is a tax that could happen. And so this malleability was a problem for people trying to do new, cool things.

So how do you fix this? Any ideas? Non-malleable signatures? So the one we did for the first homework. Does anyone have an idea about why the lamport signatures were non-malleable, like from problems at one? Yes, it was right. But yeah, they weren't. There's no randomness for one. I'm pretty sure if you flip any of the bits it's not going to work. So lamport signatures are an example where, yeah, it's non-malleable. You can't produce multiple different signatures on the same message. So that's good.

The thing is many useful signature schemes are malleable. So to just say no, you have to use a non-malleable signature scheme, it's not a great answer to the question. I'm pretty sure there's some weird malleability stuff in RSA. A lot of the systems have randomness in there and they're malleable.

So an idea that I had like 2014 and I was sort of going for was just don't sign your inputs at all, only sign your outputs. So you don't actually specify where you're sending money from in your signature. You do have to still specify in your transaction because people need to know, but you say I'm only going to sign off on the outputs. The endorsement of my inputs is implicit because the keys match. So I don't actually sign off on which key I'm sending from to something that's redundant. You know I'm sending from these inputs because the keys match up and the signature's valid for this key.

I really like this idea still. I think it's really fun. You can do a lot of cool stuff with it. It's also dangerous. It allows signatures to be replayed, which is sort of one of the big points of having utxo's because if you send two outputs-- I have address one. I send two outputs. So I've got output one, output two, and this one has five coins, this one has three coins, and they're both

the same address, both the same public key, and then I want to spend them.

And if I use this sort of signature scheme where I don't actually sign which input I'm spending from, it can be used on either. So maybe I'm not aware of this 5-1 yet, or it just hasn't happened yet, or I haven't seen it, and I say, yeah, I'm going to make a signature sending three coins over here and then someone can malleate the transaction without touching the signature, and pointing over here, and the signature wouldn't apply to either.

And then this is a really good deal for the miners because now I'm spending five coins and only outputting three and the miners get the two coins difference. And so that's pretty dangerous. And also, even if they're the same I just say, hey, I'm sending three coins to you and then as soon as the receiver sees this output, oh, it'll also work here. I'm going to take another three coins. So this is mitigated by not reusing addresses, but people reuse addresses. So it is dangerous.

I think in the context of multisig you can reliably say like, OK, we're not reusing addresses because these addresses are the combination of multiple people working together. But it would allow really cool things where you could sort of work backwards, compute a public key that you could prove no one knew the private key to, but you could still sign with it. Like really weird crazy stuff.

Anyway, people were still talking about it a week or two ago. Like, oh, we could do these cool things with it. But it's dangerous and so it's like we're not sure if it's worth it. OK. Any questions about this transaction malleability so far?

OK. So any ideas of what you actually do to fix malleability? Nobody? OK. we'll find out in one minute. Segregated Witness. I don't think it's a good name. Separate signatures would be a much easier to understand name. So Peter Wuille who is really good at bitcoin and makes all these cool things, he's not the best at naming things. Makes lots of cool stuff, but just makes whatever weird technical name.

So it's a pretty straightforward idea. The idea is when you're signing a transaction you hash a bunch of data design, but you don't include the signatures in the data you're hashing to sign them because that wouldn't make any sense. You can't. Do the same thing when you're referring to transactions themselves as txids. So in the exact same way that when you're signing, you hash the data without the signatures. When you're pointing to a transaction to say I'm spending from there, also don't include the signature data. Just take the hash of the data

without the signatures.

Yeah. You just sort of have this pointer. You've got a pointer of previous input and you've got the outputs, but the signatures aren't in there. So the idea is the signature can change and the transaction ID doesn't.

But what about backwards compatibility? So this is a great idea. Why not go for it? But how do you make it backwards compatible so that old software can still work with it? This seems like a soft fork is I'm adding new rules to the system I'm putting further restrictions on. This seems like just a change. It seems like, look, I'm now defining something in a different way. I'm removing the signatures from the txid.

How do we make this not appear to be a hard fork? Hard fork's easy. You just say, look, we're changing the entire system. From now on txids don't have signatures. So any ideas how you do this in a backwards compatible way or just give up hard fork?

AUDIENCE: Adding restrictions that screw with [INAUDIBLE].

PROFESSOR: So you can't change old transactions, but having both at the same time is tricky. So the idea is it would have been easier to start off this way. If Satoshi had just started this way, it would have went great. He didn't think of it. It wasn't a super obvious thing that-- so you can do it as a soft fork. The way you do it is you make new outputs which don't require any signatures at all and then you just don't have any signatures. This seems kind of silly.

Signatures are pretty important, otherwise any arbitrary person could just take all the money. But you redefine things in a way that new people know about and old people don't. So this is actually what a segwit output looks like. The output script is just a zero and then a pubkey hash, and then the sig script, the field for a signature is just nothing. You just don't put a signature there. And then when you're running the stack you end up with a pubkey hash on the top of the stack, which is some number and the interpreter looks at a number that's non-zero as true. Like in C or things like that.

And the bitcoins move. It's great. Someone was joking that you could potentially make this into a hard fork, because what if the pubkey hash was zero, and you found a pubkey that hashed to zero and then you signed signed with it and then segwit would accept it but old nodes wouldn't. Actually, that doesn't work, but it's sort of--

Anyway, if you're running the regular bitcoin software, you see this and no signature, and you're like yeah, this doesn't need a signature. It's just a hash. I don't know what this is. Fine, the coins move. It evaluates to true. But the new version of the software sort of adds a restriction to this kind of output. It says, look. If you see this, this is a template.

This doesn't actually mean put a zero on the stack and put a pubkey hash on the stack. It means something else. Now, it means this is a pubkey hash and look for a signature. But look for a signature in a different place. Don't actually put it in the place you're supposed to put signatures, put it in a new place. And don't tell the old software about this place. We add a new field to the transaction inputs.

It's sort of in the inputs, but they put it at the end. It's kind of weird. Logically, it's in the input. It's the same, but physically it's not, which is silly. I don't like that aspect of it. But the idea is there's this new field in the inputs called the witness field, and in cryptography, witness sort of means signature in this case anyway. It's a little bit more general.

But the old software never sees it. So the idea for here's the old transaction format. You've got your tx id and index, 36 bytes sort of pointer to what you're spending, and then a signature which is 100 bytes, and then this stays the same. And the new tx format. The idea is, yeah, the signatures field is still there. You just leave it empty.

So you're not putting any signature. It doesn't look like you need to put a signature to the old software. And then you have this third thing, which is witness, which is the same as signature basically. Slightly different format. And technically, they put them all together and put it at the end, which is kind of annoying. But anyway, logically this is how they do it. They make a new version of the transaction format.

So the old version looks like empty signatures. The new version looks like here's this useless empty signature field, and here's where the real signatures are. And you omit this to the old nodes. So when people ask for witness transactions, when people know about this new system, yeah, you give it to them. So they say hey, yeah, I'm hip to this new segwit thing. Give me a segwit transaction. And you're like, OK, here's the witnesses at the end.

But when they don't seem to know about this and they're running older software and they say, hey, just give me the transaction, you give it to them without the witness at all. It still looks valid to-- either one looks valid. However, the new people, they know that if you see this, it does not mean push a zero, push up pubkey has. There is a new rule that no, this is a template. This is

segwit.

I need a signature, and I need it to be in the witness field. So if a new node gets a transaction without a witness that they know needs a witness, they will declare it invalid. But the old nodes won't be able to distinguish. They'll say, well, it looks like no signature is needed here. OK. So you're sort of tricking the old software into accepting things that they shouldn't actually accept in some cases.

There may not be a valid signature that goes into the segwit transaction, but old software will still think it's OK. So this is how you make it into a softfork. It's kind of ugly. But, yeah. Do you have a question?

AUDIENCE: Yeah. Is this still implicitly? So when the signature is zero, [INAUDIBLE].

PROFESSOR: No, because it's based on the output script. You could make you a different output script that would have a signature that no signature requirement, and it would still work even with this new system. So it's just based on-- we changed the definition of an output script. So have this sort of template. You can still do weird-- like you could put without a zero in front.

You could put just a pubkey hash, and that's not defined in segwit. That's not defined anywhere. It would just be, OK, yeah, it evaluates to true without a signature. Anyone can spend it. And you could do that-- that would have to be a non-segwit transaction. The only way to use a segwit transaction is to have the special format for the output script. Any other questions about network stuff?

Yeah, and this solves malleability in a pretty good way. For the old software, the old nodes, well, they can't change the signature because there isn't one. There's nothing to malleate. And from the new node's perspective, yes, the signature can change, but that doesn't affect the transaction ID. Both old and new nodes still agree on the exact same transaction ID. The transaction ID does not include the witness field.

So when you're calculating a transaction, you include all this for backwards. And if there's this actual signature there, that gets into that the txid. But if you're using empty signature and only using witness, then it doesn't get into the txid at all. So both old and new software agree, and that's important, because if they didn't the merkle routes would look wrong. You take all the txids, put them into a merkle route, put that in the header. And that's really important that everyone agrees on that.

So they do work together, So that's cool. So this is kind of interesting. You've got two different old version, new version operating at the same time on the network. And they agree on a lot of stuff, but they also sort of disagree on some things. So they agree on outputs of the transactions, and they agree on which inputs there are. But they have a slightly different view of what these inputs are. Some of them think, no signatures here.

Some of them think, yeah, there's a signature here. That's weird. They don't agree on how things got spent. What are some other things that these two different classes of nodes would not agree on? Any ideas? So you understand how they see different transactions. What are some other aspects that may be sort of interesting for this consensus system that we have different views on? I forget what I put. I put two things. Any?

Hint. Biggest argument since 2010 in bitcoin. What do these two different classes of nodes not agree on? Yeah.

AUDIENCE: Size?

PROFESSOR: Well, the transaction size. Yeah. So they both see two different transactions. One of them sees it with these signatures, one of them sees it without. They don't agree on how big the transaction is. They agree on the txid. They agree on where the money is going, where it's coming from, but they have completely different views of how big this transaction is in terms of number of bytes.

So this is really interesting, For many, many years since 2010, everyone's been arguing. And one of the big aspects of, oh, if we want to increase the block size, that's a hard fork. Everyone up to now, we're enforcing. The block size must be one million bytes or less. There's no way around that, right? You can't just increase it.

We've got this rule. You're breaking that rule. This is a sneaky way to break the rule but still not tell people you're breaking the rule. Say, OK, I'm enforcing a rule that there's one million bytes. As far as I'm concerned, there are less than one million bytes in this set of transactions. The new nodes know, yeah, there's more than one million. There's like two million bytes in here.

We just didn't tell the old software about all these extra bytes. So this is kind of an interesting thing you can do. So you can increase the transaction size without telling the old nodes. So yeah, the old nodes don't see the hundred something bytes with the pubkey signature. So they

see transactions that are much smaller. Around half the size-- depends, but half the size ish.

So those bytes, they won't count those bites towards the one million byte block size limit. So this ends up being a soft fork that allows you to increase the block size. In a kind of sneaky way, right? The old nodes don't think the block size is increased. They think it's less than a megabyte, and they also think, this is weird. I haven't seen any signatures for a while.

| seems to be using these transactions that don't require signatures, and somehow everyone's getting along and not stealing each other's money despite the lack of a need for signatures. But these are not intelligent people. These are software programs, and it'll just run. And it'll, OK, yup, yup, yup. This evaluates to true. So it's kind of cool. Block size entry softfork.

However, you Institute a new rule with segwit. You don't want to just say for the new rules, we don't count signatures towards the one megabyte limit, right? You could do that, but then people might spam signatures. Let me make a giant signature or some kind of like 50 out of a million pubkeys thing and spam the network, and then it will still be under a megabyte of non witness data.

So yes, so now I've got two classes of data. You've got all the data that everyone sees, and all the witness data that only the new nodes see. So what they did is they said, OK, the witness data still counts towards that limit. But each witness byte counts as a 1/4 of a regular byte. OK, kind of weird, but yeah.

So in practice in the software, what they do is they say, OK. We multiply the non witness bytes by four. So every byte in the outputs and every byte in the txid input things counts as like four bytes. And then, the witnesses just count as one regular byte. And then we now say, OK, the new block size is four million bytes.

But four million weight units, because they're sort of, OK, we've got different weights for things. This actually makes sense, because the utxo set is what you really want to minimize, that database we keep updating every block. And the signatures don't go into the utxo set. So the signatures you don't actually have to store on a fast, low latency storage.

So in a very real sense, the signatures are sort of OK to make bigger. They don't really cost as much to the network to store. So having this discount where you say, OK, the signatures, you can have a bunch of them that doesn't really count as much. But the outputs we really need to minimize. So this one fourth is somewhat arbitrary, but there are some calculations and a little

handwaving. But it's like yeah, this is about what it should be to try to balance things.

So the end result. If you have this discount, you can put about 80% more transactions in a block. You get about 1.8 megs. It depends, right? It depends how big your signatures are. So the maximum would be you have a block that has one transaction with just a giant signature that's like almost four megabytes. And the old software would see this block as being really tiny, like 100 something bytes. And the new software would see, oh yeah, this block is almost four megabytes.

But that's sort of the extreme case. I remember generating some like 3.7 meg transaction blocks and testing that awhile ago just to test it out. It works, but in practice you're seeing about this. In practice today, as segwit has been seeing more adoption, you see like 1.3 megabyte blocks. Not everyone's using it. The idea is it's backwards compatible, but you can still use your old software.

But it seems like more and more software is using this. You get a discount on your fees because your transaction seems to be smaller. You can fit more of them in a block. So that's kind of cool, and that's sort of an incentive to use it. OK, other thing you can do. You can commit to signatures. This is a little tricky.

If the signatures aren't in the transaction ID, then they aren't in the merkle route, right? So there's nothing really committing the signatures into the block chain. And this would actually work. You could say, no, I have a signature. I'll give it to you, but it could change. It could be maleated, so it could be weird, though. You could agree on a utxo set, but you could disagree on how exactly you got there.

So one example would be multisig, where there's two of three multisig, Alice, Bob and Carol. Two of them need to sign. And then on my computer, it says that Alice and Bob signed, and on your computer, it says that Alison and Carol signed. That's weird, right? For accountability. If we want to know who exactly endorsed these things, we might disagree on it. There would be no canonical here's the blockchain, here's who signed.

The transactions themselves would all still be the same but not the signatures. So that's kind of weird, but it also seems like well, maybe that's part of the price you pay for fixing malleability in this way. If we're not putting the signatures into the thing that gets committed to in the block chain, then yeah, signatures can change. So anyway around this?

It sort of seems like yeah, that's the trade off. Sneaky way around it? Sneaky fun? No? You know. OK, so what you do actually, you commit the signatures but in a weird way. OK, so here's the regular old merkle tree, right? This is the merkle route that you put in the header. Here's all the transaction IDs, and so you make these intermediate hashes. This is the hash of these two things concatenated together, this is the hash of these two things concatenated together.

Now, if the txids don't have signatures, there's no commitment to the signatures in the top hash. What you do is this. You say, OK. I'm going to make these new witness txids, hashes of transactions that do include the signatures. In practice, you could just make a hash of just the signatures. That would also work. They just take the whole thing.

And now I've got this other reflected merkle tree kind of thing, where OK, I take the hash of these two witness transaction IDs, put it here, and this one just drops down. It's another merkle tree, and then you get a root for all those things called the witness root. And then what you do is you put the witness root in the coinbase transaction.

Put in an opp return. And the idea is the coinbase transaction doesn't have any signatures anyway, right? So you can put it in there. You don't need to include the transaction zero in this witness tree. Wait, they do though, right? But maybe this is slightly inaccurate in that I think they actually do make a witness txid for the coinbase transaction, but they define it as being zero or something. I think-- I don't remember.

So it's weird, right? But you could do that. They define a zero, or they let you pick anything you want. I would have to look at the code. But anyway, the basic idea is for these anyway, you take the hash of the whole thing including the signatures, put it in the witness root, put the witness root in the coinbase transaction, and the coinbase this transaction gets in to the merkle root. So you are committing to all the signatures but on the block level, not the transaction level.

So in the case where I think Alice and Bob signed. Oh, I think Alice and Carol signed. You can have those two transactions floating around on the network, and they have the same txid. And so who knows which one's getting into a block? They look almost the same. Some of the software won't be able to pick between them. However, once it gets into a block, one of them will be committed to.

It's like, oh, ended up being Alice and Carol. Those two signatures actually got into the

blockchain. However, you could prove, hey, no I had this Alice Bob signature, but then it never got into the blockchain, and maybe you made it after the fact. It never gets committed to. Yeah.

AUDIENCE: Also, a bunch of pool software just doesn't always do this.

PROFESSOR: A bunch of pool software doesn't do this? What you mean?

AUDIENCE: It's the responsibility of the pool software to make this construction, but [INAUDIBLE]

PROFESSOR: Have it implemented as in they just don't support segwit?

AUDIENCE: No, so they do the first part, but [INAUDIBLE] segwit support.

PROFESSOR: OK, but wouldn't that just not work? How--

AUDIENCE: It works, because-- [INAUDIBLE]

PROFESSOR: But to the new software, if you don't have-- so segwit is the software, right? You say, OK, we define these new transaction types. We define this template where if you have a zero and then this pubkey hash. It also says, I require the coinbase transaction to have this output that says, op return aa9c whatever this little four random bytes, and then I'd require it to have the witness root in here.

AUDIENCE: I'm guessing they just don't include segwit transactions?

PROFESSOR: So I've seen that a lot. Yeah, so a lot of--

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, a lot of the software says, I'm not going to do this. So the other thing that's nice-- segwit transactions to old software look non-standard. So I mentioned before that there's standardness rules where, this looks weird. I'm not going to mine it. I'm not going to relay it to my peers, but if I it in a block, well, OK, fine.

So segwit transactions look very non-standard. It looks like there's no signature. That's weird. There's this zero. What's going on? So yeah, you can you can still run a miner and just not even know about segwit. It's a little dangerous, because you might see a block that is segwit invalid, but you wouldn't know it and so you might try to mine on top of it. So there are some

risks, but in general if most people are doing the right thing, you could still mine without knowing about this stuff.

So any questions about committing to the signatures? What else? Oh yeah, so you've got this upgrade path. That's kind of cool. So it defined zero pubkey hash as hey, this is now pay to pubkey hash, right? Interpret this weird template as the regular hey, verify this signature. It also, when segwit softfork happened, redefined a whole bunch of other templates like this. So one and then some data, or two and then some data.

Just put a number, and then put a bunch of data. All of these are defined as future upgrades. So if you see a three block of data, you now say, yeah, OK. I know that's segwit version three. My software will maybe pop up something saying hey, people are using segwit version three. You're only aware of segwit version zero. But you'll consider it non-standard. You won't relay it. But if it's in a block, yeah, sure.

And you don't require anything about the signature. You'll just say, yeah, whatever weird witness data you provide for these outputs, I don't know how to interpret them. I'm just going to let it all go through. What that means is-- there's no witness needed. If a witness is provided, you just ignore it and you think everything's fine. This allows easier upgrades.

You have 16 new versions to upgrade to. Yeah, you don't require any specific things about this, so you can make new scripts, you can make a completely different script interpreter. You could say, OK, we're going to port EVM to bitcoin and disable some of the op codes that don't apply, and have that kind of thing. Have new smart contracts.

So it's kind of a fun, like yeah, we will-- and it's a nice, easy upgrade path. You could have multiple different things, things like that. The code will be easier. Don't do it today. You could construct an output that's a two byte and then your pubkey and send it out there. It will be probably stealing by miners, because everyone else's node will say, yeah, I don't know how to interpret this yet. There's no rules about this yet.

OK, let me show you some segwit stuff I looked for. OK, so there's actually nested segwit. There's an an ugly-- I didn't like it, but-- this is like somewhat designed by committee E. There's also-- this is 2016, right?

AUDIENCE: People lose so much money on segwit two years ago.

PROFESSOR: So the other thing I would say with this, I was like, OK. You've got this witness txid. And I

remember people working on segwit and I said, hey, why don't you make the transaction IDs a merkle tree of the inputs and outputs instead of just the hash of everything all together? Then, if you had a really big transaction, you could prove that an input had been spent without sending the whole transaction over.

And I thought that was a cool idea. And then when I talked to people, they're like yeah, Peter Todd already said that like three weeks ago. And whatever, we're not going to do it. It's too late. We already coded stuff. Oh well. And that's the fundamental aspect of segwit. You can't really upgrade that in the new script versions, so whatever.

There's also still a hard rule on transactions themselves being less than a megabyte, I think. So it's not a huge deal, but it would have been cool. Another thing is there's actually a way-- so there's no address defined for this, right? Address is mapped to output scripts in all the software. And so when you say, OK, I'm sending into 1aeecc or whatever, it knows how to interpret that address, build the 20 byte pubkey hash script, and send to it.

And vise versa, right? So from the address, you can get this output script, and from the output script you can get an address. So when an old software sees this, it's just like, there's no address. I don't even know what that is. I've never seen that. And so people worried that oh, it's going to be weird. People are going to have to upgrade to even send to people using segwit.

So it's backwards compatible, but if you want to say, hey, send me some money at the segwit address and then they can't. And so you say, OK, fine. Send me the money with a regular address, and then we still have this malleability problem. And then I have a wallet that supports both, and I can move money to my own addresses, and it's kind of ugly.

So they made this nested address thing, which I don't like, because then it actually has both. So you've got a signature and a witness. And the signature is not a real signature. It's just pointing to the witness. It's really ugly. There's a bunch of weird stuff in the segwit code that I'm not super into. I don't have to use it though, right?

That's the beauty of these permission-less innovation kind of systems. Like ew, I don't like that code. OK, I'm not supporting it. OK, so here's one that's nested. So I was just randomly looking through a block. Here's one, and you can see it's like, OK. The outputs are probably also nested segwit, and the input has got both a script sig and a tx witness, right? A tx input

witness.

A pure one is this one f7. OK, so you can see-- oh wait, am I not running-- what version am I running? I think I'm running to 15-1 still. So I'm not seeing the address. There's a new address format called beck 32, bech 32, which will turn-- so it's zero and then a script hash. Zero and then a pubkey hash.

It says, witness, version zero, key hash. There's also an address associated with these. I think this version of bitcoin CLI does not show it, but the new version does. So I think if you guys have version 0.16.0, it will show, here's the address. And then you can see in the single input for this transaction, there is a tx in witness. And there's no scripts.

There's a script sig field, and it's just empty. There's no actual signature traditionally. There's instead this big thing. Here's the signature, and here's the pubkey being revealed. And then it also says, OK, here's the txid without the signature, and then here's the hash or witness transaction ID. The hash of the whole thing including the signature, and they're different, right?

Also you've got size so this is actually 235 bytes, right? Because you're including the witnesses. And then, v size, which is virtual size. This is how big it looks to old software that doesn't know about segwit. So the new software, this knows about both. The actual size or witness size is 235, v size is 153. So yeah, it's not quite 50%, because this one has two outputs, and the outputs don't get any smaller, and the input just gets smaller.

And then, size, v size, and then you can see what block it's in when we get the coinbase transaction. OK, so the first transaction in the list is going to be the coinbase transaction. And I can get that one. And yeah, the coinbase transaction has a different txid and hash. Its size is 259, its v size 232. Coinbase has whatever random data they want, and there's the actual output, which is sending to this address, and it's sending 12.79 coins.

And then, there's this zero value output. So you can have an output that's got an amount of coins set to zero. It's still OK, and it's got this op return. And the op return starts with aa21a9ed, and those four bytes mean here's the segwit commitment. Here's the witness commitment to the segwit transaction hashes, the root of all those. And you have to have that in order to have a valid segwit block.

And so then we can-- this is segwit in action. I think most blocks now will have that. So there is size and v size, right? And that makes sense. But then you have strip-- no, v size is not size.

It's really confusing. And size, weight, height, like what? So size is-- I don't actually know. I think size is interpreted the same. This is the actual number of bytes for this block.

Weight is you multiply all non witness bytes by four, and you leave all witness bytes as weight one, and that has to be less than four million. And you can see here, it's just under four million. And then stripped size is the size that old nodes see. Yeah. Pretty sure. Anyway, so it's kind of confusing. One of the biggest problems in bitcoin is names, where it's like, wait. Script pubkey, and script sig script? Like, what?

All these terms and names are really confusing, and it's sort of getting worse. So yeah. Also, there's no v size here. I think this is actually v size. Anyway, so that's how segwit works in the actual thing. But it's nice, because now you can reliably spend from things before they're confirmed. So segwit is cool. Fixes malleability. Increases the block size.

Oh, it does a whole bunch of other stuff, too. OK, so one of the aspects that it fixes. When you're signing a transaction, let's say you have five inputs. Each time you sign, you need to hash the whole transaction, because it's slightly different, right? You zero out all the signature fields, but in the signature field for the one you're actually signing, you don't zero it out. You put the previous script there.

So it's slightly different. It's totally redundant. There's no reason to put it there because it's already in the txid, but you change things around a little bit. So the idea is, I'm going to put a signature here, I'm going to put a signature here, put a signature-- all five of them. Each time I put a signature here, I hash the transaction to get a slightly different thing to sign for each one.

It might not jump out at you, but this is actually o and squared, which is bad. Because the idea is, as I extend the number of signatures required in a transaction, the number of inputs in a transaction, the amount of data that needs to be processed goes up with the square of the number of inputs. Because I had an input. Now, the total size of the transaction gets bigger, so each time I sign, I need to take a bigger amount of data through my hash function.

Also, the number of signatures gets bigger. Or the number of inputs. So this is in squared. It seems fine, right? You never notice, except when you do. So there's pathological block. There was one like 2015 early in the year where some miner was like, I'm going to make this block that's one giant transaction with thousands and thousands of inputs. And a lot of software choked on it, and it took gigs of RAM to process the transaction, and things like that.

So that was bad. Just in general, if you have a lot of little dust outputs, if you're trying to aggregate them into one big-- I'm going to have 100 inputs and one output, it takes forever to sign. And it also takes forever to verify. So it's pretty bad. I remember sort of a silly story. Tim Draper's coins. He had all this dust. And it was nerve wracking, because it was way more money than I'm going to make in my life. And moving Tim Draper's coins to somewhere else.

And the software by default just swept all the inputs with that wallet controlled. And they were looking at me like, why doesn't this work? Is it frozen? I'm like, no, I'm not trying to steal the money, guys. Because everyone was sending all these little outputs to Tim Draper's 30,000 coins or whatever, because he's-- and then when he tried to spend it, it took five minutes to sign.

AUDIENCE: When people use P2 pool, the software really struggles with this.

PROFESSOR: Yeah, so it's bad. Any o event squared, this is sort of a bug. Segwit actually fixed this. The way they do it is they say, OK, we sort of pre compute these three intermediate hashes. Take the whole transaction. This is sort of the global transaction data, and pre compute these three things. And then for each of the inputs, add another thing. Here's this input specific.

So this is global. It's the hash of all the tx ends, the hash of all the outputs, the hash of this. And then here is that the input specific. Input specific. And then hash all these things into one thing and then sign that. So the idea is it's o of n in that you compute these three and then you sort of go down and keep changing this for each one. So that saves a lot of time. It's a much nicer-- oh, you also put in the amount being spent in your signature hash, which is also redundant, because that's committed to in the txid that you're sending.

But it's really nice for hardware wallets, because a lot of times hardware wallets are essentially presented with here's a hash. Sign it. And it's a very small system. It's a little chip somewhere, and it doesn't really know too much about bitcoin. It's just, here's a hash. Sign it. OK, and they don't know how much they're spending, so there could be attacks on hardware wallets, where they get a hardware wallet to sign something where it's actually moving too much money.

So it's nice to be able to have the actual amount. So there's a bunch of stuff like that. It was a giant grab bag of all these different little fixes, things like that. Fixes malleability. It increases the block size. Does all these other cool things. People didn't like it. I never really understood why.

AUDIENCE: For the reasons you've been telling everyone about?

PROFESSOR: All these reasons? Wait, these seem like good things, right?

AUDIENCE: Well, yeah, but [INAUDIBLE]

PROFESSOR: Oh. No, that wasn't what-- it wasn't like people were like, oh, here's some little things I don't like about it. Because that was what I said. That was like what everyone working on Bitcoin was like. No one thinks it's perfect. Everyone was like, oh, but this thing is weird. Why did you do that? Or why didn't you put this in kind of things.

But no, the people who didn't like it really didn't like it. There's still a bounty on [INAUDIBLE] head, right? There's death threats. Someone's like, I'll pay someone to kill this guy. It's all, this is going to destroy Bitcoin, that segwit isn't bitcoin anymore, because there aren't any signatures. It's like no, signatures are still committed to, just in a different way. You have to build this other tree.

So lots of weird conspiracies. I don't know. It became this really sticking point, and so that sort of led to Bitcoin Cash. The whole idea is segwit is bad. We're making Bitcoin Cash. And Bitcoin Cash forked off before segwit activated in the main network. Interestingly, Bitcoin Cash uses this. So they took a bunch of the code from segwit, because this is a really good improvement to signing that Bitcoin Cash used, but they didn't like segwit.

Yeah, I'm still not like-- I don't know. There's problems I have with it, too, but it's an upgrade, and it's cool. I think a lot of it was people wanted a hard fork, and this was a softfork. And so there's backwards compatibility, and they wanted to show that people have more control over bitcoin than they maybe do. It might never be possible to have a hard fork to get everyone on board to really switch. So who knows.

So yeah, it was interesting. It took forever, and that was the last change to the bitcoin code in terms of consensus code. And it was initially announced late 2015 in Hong Kong, and then all of 2016 it never-- so it activated in August of last year. And now you can use it.

AUDIENCE: People had big interest in stopping it, though. At one point they were spending hundreds of thousands of dollars a day to stop it from activating.

PROFESSOR: Yeah, so on your vert coin, you're like, I'll just take the segwit code and activate it, and like

cool. And then people tried to stop it and spend a lot of money to stop it. OK, I want to say unclear why, because I don't know. It's sort of weird. There's a whole lot of opinions.

One theory is that this breaks some mining chips optimizations. One of the optimizations-- it doesn't work with a tree of height two. But if you have a really tall tree, you can swap txids, or you can swap intermediate nodes of the tree and you'll get a different merkle route. So you can see-- so it doesn't work here, because this has to stay in place. But in many cases, the order of the transactions is arbitrary.

So I could flip these two. It's still valid. So what I might do is say, OK. I have this merkle route I'm mining, and then I want to flip these two, calculate a different merkle route, and mine. And there were some chips that maybe did this and had these kinds of optimizations. There was also a patent on it and all this weird stuff going on.

It doesn't break, but it essentially loses the optimization if you have this. Because you're saying, OK, I'm going to have this big tree. I'm going to swap something near the top, and it only has to recompute two hashes to get a new merkle route. However, if I now have this mirror image witness merkle tree underneath, if I say, OK, I'm going to swap this, I'm also swapping all these.

And I have to recompute this. Maybe I can swap some of it, but I have to recompute what this. This is going to change just as well. And then I have to put that in here, and this is going to be at the bottom. And then, I'm going to have to recompute everything all the way up to the merkle route. So this was called AsicBoost, and then there was a post-- Greg Maxwell posted this sort of like, you guys, like accusatory mail on the mailing list last spring saying, look.

We were trying to figure out a way to break AsicBoost, because we think miners have this patented algorithm that optimizes and it gives a 20%, 30% speed up. And we're worried that the patents will make one miner, have a monopoly, and everyone else won't be competitive. So we're trying to think, is there a way we make software to prevent this from this optimization?

And then once they tried to look at it, they were like, oh, wait. Segwit does that. We want to make it costly to swap things in the tree, and segwit does that. Oh, so basically, we're good. And then he was like, oh, wait. Maybe that's why all these people hate segwit. Maybe this is these miners who have billions of dollars worth of equipment with these optimizations in it, which would be rendered unusable by this new software change, maybe they're trying to

prevent it from activating.

It's a theory, and the mining companies said, oh, no that's a bunch of nonsense. Although, the way they said it was sort of suspicious. They were like, yeah, we put circuitry in our chips to do this, but we never used it. That's strange. So who knows. But that's one theory. I'm not sure how much I believe that's the real reason, but yeah.

AUDIENCE: but if they want to calculate Merkle roots in bitcoin, don't just-- order all of the transaction fees by transaction ID?

PROFESSOR: You can't, because the order matters. Because when you validate, this transaction might create an output that this transaction spends. And so if you swap them, so if you didn't have intra block dependencies, then it would all be arbitrary and you could put in ordering. But there are intra block dependencies, and so the order does matter.

In many cases, it doesn't. In many cases, these are two separate transactions. You can swap them. But the software does use the ordering. And there's all sorts of other things that would be better. What I would want is prepend or append the height at each stage of the merkle tree. That would have helped me out for some things. Because then, it's like you know, since you're at the bottom just put a zero at the end of each hash. And then when you get up here, put a one at the end of each hash.

Doesn't really change anything. But one problem is what if I request-- so what I want to do in my software. I want to request all the transaction IDs in a block. I don't actually care about the transactions. I just want to see all the txids. Like this. If I get rid of the head 20, I get a giant list of txids. The thing is, what this let me do is to look for transactions.

If I have a txid I know I'm looking, I can say, oh, I can look for it in here. The problem is, what if the person I'm asking is giving me this instead of this? I won't know. They all look like random numbers. If I do the merkle tree algo, I'll get to the merkle route. That's good. But I don't really know that I'm at the bottom. It's OK if I'm running a full node and I actually download all the transactions and look, and OK, it works.

But to have a way to say, hey, give me a list of all the txids and I can verify that it's correct, I can't do that right now. There's ways around it. But it would have been nice if then they appended a zero or something. Or even, all you have to do is just append something at the bottom row or just append higher or something. Then, it would've been kind of cool. It

would've been easier for me.

But oh well. And there's people who've written about this. Yeah.

AUDIENCE: Did James say that's pool operators are leaving off the [? whipper? ?] And if so, does it weaken the whole system?

PROFESSOR: I think what they really do is they just don't support segwit. So I've seen, especially--

AUDIENCE: [INAUDIBLE] it's expensive but then they--

PROFESSOR: Yeah, they say they're going to support it, and then they don't. So they sort of flag their transactions, yeah, segwit, and then they haven't actually upgraded their software, so they can't use it. They can't mine it. So you see this a lot on TestNet as well. If you're making TestNet segwit transactions, sometimes they just don't get confirmed for a few hours, because all the blocks that come out don't support it, and so they won't use it.

AUDIENCE: The badly written pool software, if they use segwit supporting full load with it, it will give them segwit transactions, and they'll try to include it but it won't do this, so--

PROFESSOR: So it's invalid. Yes, so it's invalid.

AUDIENCE: I guess my question is does it weaken the security in the system if for six months they're not supporting this?

PROFESSOR: No, no. It hurts the usability. If I want to use a segwit transact-- but as me running a segwit compatible node, I require signatures. I require all this whole construction. If you make something that looks like it spends the segwit transaction without this, I just reject it. So security wise, it's fine. Yes.

AUDIENCE: I think it might be important to note that the way that these things are designed, and in particular that softforks are designed, is that anyone who doesn't update the new functionality can't hurt the security of the new functionality. That's sort of part of the design process.

PROFESSOR: Although, their security might get hurt. Not a ton, but yeah. If you haven't upgraded, you might see these segwit transactions, and--

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, they look weird, but you're like, OK, fine. But you can't actually verify the whole thing.

Given an invalid and a valid segwit transaction, the old software can't distinguish but the new software can.

AUDIENCE: That's even though the pool operators, whether there's six or eight key pool operators, might not be supporting the witrootsub

PROFESSOR: If they don't support it, you have to wait until someone that does support it mines the block. So if they try to support it and support it wrong, you ignore them. You don't use their data. You don't use their block.

AUDIENCE: you just want segwit transactions stay in the node pool a bit longer.

PROFESSOR: Yeah. So I think in bitcoin now, it's OK. TestNet is kind of weird, but there's segwit.party, and you can see what people are doing with segwit. So yeah, it's about 30. This is by transaction, it's somewhere around 30 something percent of the transactions are using segwit, and then you can see witness size percentage, block size. OK, so sometimes you got-- oh wow, I had no idea. Blocks are way under a megabyte now.

Oh, OK, well free transactions for everyone. If you want to use bitcoin, now's the time. You don't have to pay anything. That looks very different a month ago where you had a solid red line. You had to sort of-- nothing went below a million, and then you had a little bit of segwit stuff going on here. But now you've got most things are below a million. So interesting.

OK, so yeah. So that's the basic idea of segwit. And if people have any questions, stick around and ask. There's office hours tomorrow at 4:00 to 6:00 over there. Look at the homework, and next time I'll talk about lightning network payment-- I'll try to get into payment channels and see how far we get into lightning network stuff.