TADGE DRYJA: So today I'm going to talk about MAST, taproot, and graftroot. And these are pretty cool new ideas in Bitcoin. They're not really production ready. They're still sort of ideas, but it's kind of neat to look at the future directions of how people are thinking about improving these protocols. So we're going to talk about new types of scripts-- MAST, taproot, and graftroot-- and all those sort of interactions between these three things. I will define these, of course.

So so far we've talked about scripts. And you've got up codes. You've got a stack. I think with the homework people sort of got the idea, OK, I can do OP_RETURN. I can do OP_CHECKSIG, and the basic idea of how it works.

So right now in Bitcoin, it's mostly pay to pubkey hash. So most of the scripts that you'll see-- mostly, output scripts in an output look like this. OP_DUP, OP_HASH160, thumb pubkey hash OP_EQUALVERIFY, OP_CHECKSIG.

And then when you want to execute the script, you put a pubkey and a signature as your input. It duplicates the pubkey, hashes it, pushes this hash onto it, checks that those two things are equal. And this also removes those two things, leaving only the signature, and then checks the signature.

So if they're not equal-- if you try to sign with a different pubkey than the one that is committed to in this pubkey hash, it'll fail at this point. And if you put the right pubkey but the wrong signature, it'll fail at this point. Well, it'll the CHECKSIG will fail, put a 0 on the stack. And then the stack will have a 0 on it, and that'll fail. Verify means, immediately fail, if these things are not equal.

And then in segwit, they templated it. So really, this entire script became OP_0, pubkey hash, which is confusing. If you don't know what that means-- wait, OP_0 means just put a 0 on the stack, and then put a pubkey hash on the stack, and anyone can spend this.

But the idea in segwit is, yes, software that is not aware of segwit will see this as spendable by

anyone without a signature. Software that is aware of segwit knows that these two things are the same. So essentially, if you see this, turn it into this.

Take the pubkey hash and fill in these OP codes. And that's actually what the code does, for the script interpreter. It will say, oh, it matches this template. It's OP_0 and a 20-byte piece of data. Turn it into this, and then execute it. So in this way, segwit saves three bytes. Because now you have one byte there instead of 1, 2, 3, 4. You lose a couple more bytes elsewhere in segwit, so it actually tends to lose a couple of bytes. But whatever.

So this works. This makes sense. Any questions about pay to pubkey hash? It works OK?

And there's also pay to script hash. This is the other thing you'll see quite a bit of in Bitcoin today. I don't know what the ratios are. 80% pay to pubkey hash, 20% pay to script hash-- something like that. So a minority, but significant.

And for pay to script hash, it looks like this-- also sort of template-y, right? OP_HASH160, a script hash, OP_EQUAL. That's it. This also seems like, wait, all I have to do is provide a pre-image of this, and then I'll be able to spend the coins. But the script interpreter knows-- no, actually what this means-- if you see this template, if you see this format-- verify that this is equal, and then execute whatever the pre-image was, as a script.

So actually the first time I started programming Bitcoin stuff, I did not know about pay to script hash, and I was just trying to make my own transactions. And I used this. I just said, well, I want to see that I can get the script to execute right. I'll just take a random number, hash it, put it into this thing, and try to spend it. And it never worked, and I didn't know why. And then a few days later, I read about pay to script hash.

Wait, it wasn't days later. I just put an OP. I put an OP, no OP, OP_HASH160, script hash, OP_EQUAL. And then it worked. And I was like, OK, I don't know how Bitcoin works. This doesn't make any sense. I put a no OP, and then it starts working.

And then a few days later I read about pay to script hash, and it was, oh, this actually has a special meaning. This is a template where if you HASH160, a bunch of data, and OP_EQUAL, then it interprets the pre-image as a script. Does it have to be 20 bytes, probably? Yeah, I don't know.

There's actually a vulnerability here. How pay to script hash got introduced is sort of an interesting debate. There was OP_EVAL, pay to script hash-- a bunch of argument about it.

And this was in 2012, I think, or '11. And it was sort of contentious, and people argue about it.

And in retrospect this is actually insecure because you're using OP_HASH160 160, which is 160-byte hash function. It's basically SHA-256 followed by RIPEMD-160. This is not big enough. You're committing to a script hash, which is 20 bytes.

So normally you're like, yeah, whatever, 160 bits is plenty. You can have collision attacks, where if you're doing multisig with someone else, they may be able to calculate different pubkeys, so that your pubkey-- they have a collision where two different 20-byte scripts hash to the same thing. Sorry, two different scripts, hashed to the same 20 bytes. This is possible because that's only 2 to the 80 work which is still a lot. And used to be considered like, no one can do-- 2 to the 80 is beyond any human endeavor. But Bitcoin does 2 to the 80 work every day, hour. I don't know, something fairly reasonable.

So it's like, wait, 2 to the 80 is not enough. For the same security as Bitcoin, which is generally a 2 to the 128 security parameter, we would need this to be 256 bits instead of 160. So when they upgraded the segwit pay to witness script hash, this has to be 32 bytes, to eliminate any of those collision vulnerabilities.

So yeah, that's what distinguishes it. So they look the same, Right it's just OP_0, piece of data, OP_0, piece of data. But they do completely different things based on how long this data is.

OK, so that's pay to script hash. Any questions about the basic construction?

OK, I want to make it so that two of these three keys must sign, in order to spend the coins. So I say, OP_2 push three public keys OP_3, OP_CHECKMULTISIG. So the idea is two of these three must sign. So you can put OP_1, this OP_3. You can OP_3-- this OP_3. If you put OP_4, three keys, and then OP_3, it probably will never be spendable and just never work. I'm pretty sure you have to put the signatures in the correct order. So when you spend, you say, OP_0, sigA, sigC. And that's A, C. If you put sigC, sigA, it fails.

This is a fun little part, where you have to put in a 0 byte before the signatures, when you want to spend. I'm not 100% sure how that got in. It was a screw up, and we just live with it forever. [LAUGHS] If you don't put the OP_0, it doesn't work. And it's somewhat confusing. It wastes a byte, which is not a huge deal. But it's just like, what? And they didn't fix that in segwit. It just stays in there. Who knows if it'll ever get out. So segwit was an opportunity to fix it.

Yes, go ahead.

AUDIENCE: Why didn't they just update that pretty quickly when they realized you have to put an OP_0 there?

TADGE DRYJA: Hard fork. So you put this in. And you could say, wait, that's stupid. Wait, you need a 0 in front when you put the signatures? Let's get rid of that. So now the old software says, no, I need a 0 here. And then if you don't put the 0, and you just put the two signatures, I fail this script.

AUDIENCE: [INAUDIBLE] doesn't have to be a 0 until the [INAUDIBLE]?

TADGE DRYJA: Right. It could be anything. It could just be any data. And it was like any data the stack would get popped off. And then the two signatures. So if you said, let's eliminate that requirement that would actually be a hard fork, because it would be allowing scripts that were previously invalid to now become valid. And that's a huge undertaking, that has sort of only happened once in Bitcoin, and that wasn't even planned. And so it's just like, whatever, leave it. They did soft fork it then. Is it soft forked or policy?

AUDIENCE: Soft fork.

TADGE DRYJA: So they did soft fork and say, it needs to be a 0.

AUDIENCE: [INAUDIBLE]

TADGE DRYJA: Yeah, I'm not sure. So it could be non-standard, or it could be an actual consensus rule, that that has to be a 0. It used to be anything, and now it should be a 0. Put a 0 and then it'll work. But that remains in segwit as well, with multisig. They could have fixed it. They didn't want to touch it. It was just like, whatever. When you see one of these, turn it into one of these, and just do the exact same thing you used to be doing. That's a lot simpler and easier.

So kind of weird, but this works. People use this a lot, especially exchanges. Exchanges very often use this kind of construction, because it helps people manage keys. If one key gets compromised, you don't lose all your money. Usually, 2 of 3, sometimes 3 of 4. 3 of 3, 2 of 2-- all sorts of constructions where this is extremely useful. And actually, I'll be talking about more powerful signature aggregation and multisig on Wednesday. But for now, this is what we use. And, how can we make this better?

So one interesting thing-- pay to pubkey, which is not really used anymore, but used to be

used more frequently, like in the early days. If you look at transactions in 2009, 2010, this is predominant, I think. And all the mining outputs from a long time ago looked like this. And the output script was just pubkey, OP_CHECKSIG, which is really simple.

And then the idea is when you want to spend it, you just push a signature onto the stack. Then you have signature, pubkey, OP_CHECKSIG. That's it. It's really simple, and it actually saves space. Your pubkey is 33 bytes, whereas your pubkey hash script, the pubkey hash is 20 bytes. So yeah, the pubkey hash is 12 bytes smaller plus the extra OP codes makes it 10 bytes savings for the output script.

However, when you spend with data pubkey hash, you need to put the pubkey itself and the signature. So you're putting the pubkey into the network either way, eventually. So this is actually overall more space efficient, in that you're saving 23 bytes over the lifecycle of it. The output is a little bit bigger but the input is significantly smaller, so you end up saving.

Why do we always use pubkey hashes instead of pubkeys? There's a lot of different opinions on it. You know, it's Bitcoin, there's no "answer." People have different ideas of why these are better. Any ideas?

AUDIENCE: [INAUDIBLE] put their pubkey out in public before it got spent.

TADGE DRYJA: Yeah, that's probably the predominant one. And that's going to be a tricky one to try to convince people is not the case. I'll get to that. But people started arguing about pubkey hash. When someone posted about graftroot, the discussion devolved into a, wait, I don't want my pubkey on the network.

And there's some arguments why you'd want a pubkey hash, and it's more secure, but there's a lot of arguments that it doesn't actually provide any extra security. So that's a tricky one. But that is one. That is probably a big one people like. Other ideas why you use pubkey hashes instead of pubkeys themselves?

Size-- you want the outputs to be small. Also, I'm pretty sure when Satoshi wrote all this, he did not know about compressed public keys, because that stuff was not in there for a while. I think I mentioned this before. Did I mention compressed versus uncompressed pubkeys? Yeah, probably. OK, so review.

The idea is you've got this whole curve kind of deal, kind of thing-- usually like that. Your public key is a point on the curve. You need an x and y-coordinate to represent the point. However,

because it's symmetric, you can just say, well, I'll tell you the x-coordinate. You know, x equals 7. I'm not going to tell you the y-coordinate. I'll just say y equals plus. So go up, or go down. That way, I can do 32 bytes for this and 1 bit for that, at the cost of a little bit of extra computation.

So I can say, maybe y is actually 5, or something. So if I say, x equals 7, y equals 5, that's quick but it takes more space. If I say x equals 7, y equals plus, I now have to compute what y is when I see this-- which is pretty quick, but take some time. But it's a good tradeoff. Anyway, I don't think Satoshi knew about this idea-- or maybe thought of it, never put it in. So the public keys were actually 65 bytes.

You want to keep the output small, because they're in the UTF subset. They're in the database. And you need to be able to think through this and read it randomly. Like, you don't know when it's going to get spent. You don't know what transaction is going to query this. So this has to go in an actual database, key value store kind of thing.

The signatures don't have to be in any database. They're just in these old blocks, and you read from them linearly. You basically read through the whole block at a time. And high latency is fine. There's never any timing-critical reads for this.

So basically, you could store the blocks as just files on your hard drive. Block 1 dot dat, or whatever-- block 2 dot dat-- and have 500,000 files, because you basically always read them a block at a time. And you only have to read them when other people are doing IBD-- initialization block download-- and you're serving it to them. So you can prune these, if you run Bitcoin core with pruning.

Anyway, so the main idea is, OK, even if it overall increases the size of transactions, it does reduce the size of the outputs-- which is the thing we really want to minimize, for keeping the speed up. Questions about that? Cool? OK.

So similarly, same thing with pay to pubkey hashes versus pay to pubkey. Instead of pay to script hash, you could put the whole script in an output-- like a multiset. You could just have all three of your keys in your output script. And that would save space overall. Your output is 20 bytes, and then you reveal your whole 100-byte script, so you save those 20 bytes. Same amount, but proportionately less, sorry.

It is some space savings overall, but it's better to keep the output size small. [INAUDIBLE] If

you do this, does it actually work?

AUDIENCE: [INAUDIBLE]

TADGE DRYJA: OK, it's not standard, so it might not propagate.

AUDIENCE: [INAUDIBLE]

TADGE DRYJA: I wouldn't recommend this. It probably will not really work. Maybe eventually it will get confirmed, if you find people to confirm it for you, but most nodes will ignore these kinds of things.

OK, so the question for today-- what if we want really, really big scripts? So 2 of 3 multisig is cool. We can just show all three keys. And there's 33 bytes of data that don't really get used. If you're signing with 2 of the 3 keys, well, that means one of the keys never got used. It was sort of there and pointless. But whatever, it's 33 bytes.

So what if you want to do, like, 2 of 50 multisig? So here's 50 different keys. Any two of these people can sign. Maybe you're in a company. There's 50 employees. And the policy is that, OK, any of these two people can use the coins for this spending to buy lunch, or whatever. So how would you 2 of 250 multisig? Any ideas? You commit, and only reveal part of a commitment. So the cause of and solution to all of blockchain's problems-- Merkle trees. That's sort of the solution for all-- throw a Merkle tree on it, which is sort of actually what I'm working on now, for another thing.

So you commit to some kind of root, and then you reveal only part of what you've committed to. So the idea, which is from-- there's debate about who came up with this stuff-- MAST. Like Peter [? Wohl ?] was into it, but then someone else said they wrote a paper about it. I don't know. It was like 2013, 2014.

The original idea was make a merkelized abstract syntax tree. So if you have done any programming language courses, you probably recognize the term abstract syntax tree asked, which is sort of how compilers work. They make this tree.

So the idea is, OK, we'll make every node in that tree like a Merkle root, so every single OP code can be revealed and be swapped out. And it's super crazy. That's sort of overkill. So that was part of the initial idea and paper. But a much simpler way to reason about it and program it would just be something like, pay to script Merkle root. Where in my output script I have a

root, and then I reveal that there is a script that goes into that root, and then execute that script.

So for example, I make four different scripts. So this would be-- anyway, it's just four different scripts. There's an O-2 choose n, that results in four. I don't think so. But you know, this is Alice and Bob. This is Bob and Carol. This is Carol and Dave. This is Dave and Edna, or something. And any of these two people, but four different groups, can sign.

And then this is what you actually put in your output script. And then when you want to spend, you say, OK, I'm going to provide a proof. So to spend, reveal which I'm spending. So for example, I'm spending using script 3. And then I reveal a path to the root, which would be I reveal 4, and this, and then you compute.

You say, OK, hash these two things together, you get that. Hash of these two things together, you get that. And confirm that the result is equal to what was in the output script. And if it is, OK, you've provided a valid inclusion proof that script 3 was inside this root.

What's kind of cool-- in this case, you do not need the binary tree to be perfect. You can have different heights, because you're not committing to the height of the tree when you're sending to it. So the default case could be here, and then a really big tree that goes down a lot over here.

So in most cases it's going to be Alice and Bob signing. And then Carol and Dave, and Edna and whatever, are unlikely. So I'm going to put Alice and Bob here, and then have the tree sort of asymmetrically get longer over here. So that when Alice and Bob sign, they can just reveal this. And then when the other people sign, they're going to have to reveal this and this. And then I optimize my space for the most likely case. So that's kind of cool.

What else? However, there are a bunch of problems. One, OK, 2 of 50 is cool. So in the case of 250, let's just look through it. 50 choose 2-- I couldn't make the little numbers in parentheses the right-- whatever. So that's 1,225 different scripts. And if you do a regular binary tree-- balanced-- I forget all the words-- you get a tree height of 11. Which is kind of like, oh, it's so close, right? If it was 1,024, it'd be 10, but a little over.

And actually, since it doesn't have to be symmetric, you can get like tree height 10 for most leaves, and then 11 for some of them. Anyway, so your proof size is going to be 352 bytes, which is OK. And if you didn't use MAST, you can say, well, I'm just going to have a pay to

script hash.

And this script is just going to show all 50 keys, and have OP_2, all 50 keys, OP_3. And that would be about plus 3, or something, but 1,650 bytes. So MAST is an improvement. And it's sort of like, hey, I'm doing log n. And for all the different scripts, I have log n. If I actually committed to all the 1,225 scripts with each of the two keys, it'd be much bigger. But I don't actually have to, because I've got this whole 50 choose 2 in here. So I've got two things. I've got this combinatorial thing, and then this log thing. And in balance, it's like, OK, meh-- like, better, but not actually that awesome. You're going from 1.6k to 350 bytes.

It gets worse. Let's say I want to do 25 of 50. Well, that's 50 choose 25, which is around 100 trillion different combinations. The tree height would now be 47. The proof size is going to be-- wait, did I do that wrong? Hold on. 47 by 32 is 1,504. OK, sorry, for some reason I have 22 there. Sorry, I'll fix that. But that is correct. Yeah, 1,504. So 1.5k, whereas the raw would still be 1,650. These are about the same. This is not a great space savings. Your proofs are huge.

In fact, I would say it's not only not much better, it's worse. You're going to have to compute 200 trillion hashes in order to compute that root, as the person creating the output script. To verify inclusion, you have to compute 47 hashes, which is fine. 200 trillion hashes is doable, but that's going to take a while. And that's annoying. How long? Hours, days? I don't know. Doable, but not fun.

So MAST doesn't really get you necessarily what you want here, although this is kind of a silly case. None of the things really get you what you want here, because of these sort of combinatorial problems. Any questions, or oppositions if I have screwed up some math somewhere?

AUDIENCE: [INAUDIBLE]

TADGE DRYJA: So compare these two. If I show a tree height of 10 or 11, huh, that's weird. There's like 1,000 different scripts around in this tree. If I show a tree height of 47, that's really weird. Someone either was just joking around and made this weird sparse tree with all these branches to just screw around and have a height of 47, or there's 200 trillion possible spending outcomes. So it reveals information about the things that were not spent. Even though it doesn't reveal what the scripts were, it to some extent reveals how many there were.

But still it is kind of useful, right? In the 2 of 50 case, yes, this is better. You have to compute a

thousand-something hashes. No big deal. You reduce your size from 1.6k to 353 bytes. That's a good deal. Here kind of not, but this is also sort of a crazy-- there's other ways to address this, but it's pretty good.

There's sort of two, or more than two different ways to implement this. You could just say, OK, we're going to have pay to script Merkle root, where you have a template-- where it's like OP_2, or OP_3, or something, and then some data. And the idea is when you spend it, you have to provide a proof, and then the script itself. And then, you execute the script.

Another way which is kind of cool-- and people were talking about it recently-- you have this tail call recursion-looking thing. Where the idea is, if you evaluate your script and then there's two items left on the stack at the end, what you say is, OK-- whichever is bottom or top-- one of these items, the bottom one, is a Merkle root.

And then the other item on the stack is the proof and arguments-- another script. So you can sort of recursively say, OK, well, I went down into this script with this proof and then executed it. And then things were left on the stack, and it was another proof, for another Merkle root. So that's kind of cool. You can make all sorts of crazy things with that.

I don't know how useful recursion is in this case, but there could be cases. You're already in a binary tree. So my personal thing is, why not just keep it simple and pay to a script Merkle root? But there's people who think, no, this should be more expressive, like programming language-y wise.

Anyway, so that's the state of MAST. MAST has code. There's a pull request in bip, right? Yeah. So there's some people who are really into MAST. And like, let's get this into Bitcoin. Let's make a soft fork. And there's also people who are not as enthusiastic about it. And so it's like no one's opposed to it-- like, this is a bad idea, this will break Bitcoin. But there's varying levels of let's get this in, you can do all these cool things, versus eh, is this really a priority? What are we going to use with it, kind of thing.

OK, any questions about this? I think we have a quick break, and then I'll do OP_RETURN, which seems unconnected. But I will show you how this connects to these issues we have.

OK, so OP_RETURN, it was in the problem set. You basically put this OP code in the front of your script, and that flags it as forever unspendable as an output. So you shouldn't put much money into these outputs, because they are basically destroyed. OP_RETURN means you

immediately return from the script evaluation, and you return a false.

I think it used to be return whatever is on the top of the stack, which meant you could just spend anyone's money by using OP_RETURN. So they soft forked that to say, no, OP_RETURN always fails instead of OP_RETURN always trivially succeeds-- which means you just put an OP_RETURN and take anyone's money. So they figured that one out pretty quick.

So people use OP_RETURN to put data into the blockchain. Why do people do this? You can't spend it. You can't do anything with this data. But why do people do this? It's kind of cool, right? To prove it's there. You want to put some data in. And you just want to say, hey, I thought of this first, or I'm making a patent, whatever. You put data. People use a lot of OP_RETURN. What was it? Like, 10%?

AUDIENCE: [INAUDIBLE] set. [INAUDIBLE] the current [INAUDIBLE] set, if you remove, or if you don't for the unspendables, it would account for about 10% of [INAUDIBLE] data set.

TADGE DRYJA: So that's a lot of data. People were using it. There is a better way. So what if you wanted to do a 0 byte OP_RETURN? It's not quite the same. Because an OP_RETURN, everyone can see it. And in some cases, you might want that. You might want to put an opportunity so it's publicly visible. But in most cases, I don't care about someone's random OP_RETURN. Usually, what you're putting into OP_RETURNs are, like, hashes. Because you don't have much space anyway. You have, like, 40 bytes or something. So usually your space constraint so you put the hash of some document. And then someone can prove, no, look, I put the document on the blockchain, by putting the hash of it. That's a commitment to that document itself.

So I show that I had this data at some time before this block came out. That's essentially what you're proving. You're proving you knew it before a certain time, which is in many cases really useful. I mean, that's sort of the whole point of the blockchain, to prove that you knew about this transaction before a future transaction tries to double spend it. You want to file a patent, and I had this idea first-- or prove these things.

But how can we do this with 0 bytes overhead? OK, it's not obvious. You put it in the signature. And this was not obvious, and no one thought of this until a year or two ago. I think it was Andrew Poelstra, but I could be wrong. But I just associate it with him, because he worked on it. But I'm not 100% sure. And he called it pay to contract hash, which is confusing. It's not at

all the same as pay to script hash. It's not at all the same as pay to pubkey hash. But it's called pay to contract hash, which makes it really confusing. Bitcoin has really confusing names, like script pubkey.

Script pubkey does not have to have a pubkey in it. In most cases, it doesn't. It has a pubkey hash, or a script hash. It doesn't have to have anything in it, but it's called script pubkey. And then script sig usually has signatures, but again, not always. And there's all sorts of confusing names.

Anyway, here's our new contribution to confusing names-- pay to contract hash, which is not really paying to anything. There doesn't have to be a contract. There is a hash involved. So it's a weird name, as you can't actually detect the contract, you're not paying to it. But to review, I'm going to use Schnoor signatures. This still works in ECDSA. But ECDSA is weird and annoying, and it also makes more sense in Schnoor signatures, so I'll explain that. But it totally works in ECDSA.

The idea of the Schnoor signatures-- OK, this is my private key. This is the message I want to sign. This is a random sort of temporary private key I come up with. And then this is that temporary private key turned into a public key. Multiply by G to get the public key version. And then to verify the signature, the verifier multiplies both sides by G. So you get s times G. k turns in R. Little a turns into big A. This is just the 32-byte hash, so it stays the same. So the R and s are the signature, and the pub key is A, and the message is m. So that's the signing, or close enough to the signing that's used in Bitcoin.

So pay to contract hash says, OK, I want to put arbitrary data into my signature, in a way that I can prove it was there. So here's the normal equation. Now, what if I redefine k. Normally, k is just a random number. But now I'm going to redefine k. k is now j. So j is my new real random number. And then I'm going to add the hash... There's no G there, shoot. Anyway, G should not be there, but it should be there. Sorry.

So I redefine k as j, which is an actual random number, plus the hash of the data I want to commit to-- and j times G. You get the idea. That should have been there.

And-- then the signature-- it's still the same equation. I'm just saying, OK, it's this new k minus this thing. And so nobody necessarily knows that I'm doing this. If I just tell you tell you R and s, the equation still holds. It's just that k is a little special. For anyone else, it looks like a

random number. It's a random number plus this other hash-- not times G.

So it's a random number plus a hash. Which hashes look pretty random-- so it's a random number plus some other random number. No one's going to tell that anything's weird about this. However, it is unique, in that I've got this random number plus the hash of that random number times G in it, and this extra data I've put in.

So to verify it, if I don't tell you about this, verification looks the same. What if I do tell you about it, though? So normally the sig-- R, s-- pubkey is A, message is m. But the signer can prove that R is not kG-- that there's this other k. Well, it's k, but there's something else in R. I can prove that R is special after the fact, right? Everyone knows what R is. But I can then prove that something was in R. Also, never reveal k, even after the fact. That will let you figure out the person's private key.

So the idea is I can prove that R equals J plus the hash of the data in J times G. Here's where it's a little confusing, right? Because this is already a pubkey, and you're multiplying by G again, right? So it's a little weird? Anyone can compute this, right? Because I can say, hey, you already know R. I can give you a J and some data, such that j plus the hash of this data, and j itself times G, equals R. And I can't forge this. If I try to forge this, and I say after the fact-- I just made R randomly. I didn't have any actual data. And then after the fact, I want to come up and lie, and say, I put this data in to this R.

OK, now we need to solve for J. Because my proof is going to be J in this data. So I've got the data I want, and I want to figure out what J is. So J is the hash of this data. And J times G minus R-- oh, shoot. That would be the equation I have to solve, if I want to falsely prove that this data went into it. The problem is I've got this thing where it's like, J is defined in terms of the hash of J.

That's going to be hard, right? If I'm defining J in terms of the hash of J, then I'm stuck, because hash functions don't let me do that. So this is actually kind of cool, because it's a proof that there was some extra data that went in to the random R point, before the signing happened. Yeah, it had to be before the signing happened, because the signature has R in it, and S is a function of R.

And this is cool. Because if you don't know about it, it just looks like a regular signature with a regular R value. But if I tell you about it-- like, yeah, when I signed that, I actually put some extra data in here. So that's kind of better than OP_RETURN. There's 0 bytes overhead. You

were going to sign anyway. So you can put data inside a signature's R point. You can even do it with other people's signatures. There's no real private information here. If I just hand you some data, it's totally safe. Hey, here's some data. OK, here's the J point, that constructs this, and the R. It's like, OK, sounds good.

I can put someone else's data into this. You just hand them the data. They give you the proof, which in that case is just J. So it's OP_RETURN with 0 byte overhead. This is really cool. I believe this is being used for open timestamps now, as of a few-- I think, right?

AUDIENCE: I wasn't aware of that.

TADGE DRYJA: I think they got it in. I don't know. But it's also nice-- it's not like a soft fork or anything. This is just a fun trick you can do with your signatures. So you could do this today if you wanted, and then prove to people later, that, hey, I put my name into my signature.

AUDIENCE: [INAUDIBLE] the hash, wouldn't proving one essentially refer to all in the batch stuff?

TADGE DRYJA: This is direct replacement for the OP_RETURN. You commit to the root. And in their case, this data is a Merkle root. And then you can improve. So you can put a Merkle root here.

And then I prove that there's a Merkle root inside my signature. And then I also prove a branch, so that the data is in the Merkle root, which is then in the R point-- which is in the signature, which is in the block's Merkle tree, which is in the blockchain. So you've got, like, four or five different proofs going on, but they should all be working.

OK, so this is kind of cool. Any idea why I was explaining all this in terms of MAST and multisig? These seem totally unconnected, right? Any idea?

It took Bitcoin people like well over a year to see the, in retrospect, really obvious connection, and thing to do with this. But it took a year or two. And then Greg posted on the mailing list in January-- like, hey wait, you can use this for something else. And then it was like, oh shoot, how did no one think of that?

So it's called taproot, and I don't know why Greg called it that. Greg Maxwell, he's a guy with a big beard, who works on Bitcoin. He's super smart and stuff. And it uses this pay to contract hash construction. And it's the same equation, but it took us a year or two to figure out that you could do this.

So the motivation here is pay to pubkey hash and pay to script hash look different. You can tell just looking at the output, these are totally different output scripts. One of them has got a pubkey hash. One of them has got a script hash. Different is bad. We don't want everything to look different. Then, you can sort of sort things. You can try to differentiate, try to track people's coins, things like that. You want it to all look uniform, to make it more anonymous.

So one thing you could do is just use pay to script for everything. You can put a pubkey single-- essentially, 1 of 1 multisig, or just pay to pubkey hash script in pay to script hash. You wouldn't do that. You'd have that you'd have the direct pay to pubkey put into a script hash. So that like it looks like a script hash, and then the script is public key OP_CHECKSIG. It's actually not really any overhead. I think you actually save a byte or two. Nobody does that. You could. That might solve like 80% of what this is trying to solve. But it's kind of boring and doesn't use cool math, so anyway.

OK, the other observation is that in most cases scripts-- you have these sort of scripts. And there's a hidden option, which is, or everyone signs. So in the case of 2 of 50 multisig-- well, if all 50 people sign, that's OK, right? That's no worse. So if it's 25 of 50 multisig, if all 50 people happen to sign, sure, that's no loss of security there.

The idea is you use this pay to script, pay to contract hash construction to merge pay to pubkey hash and pay to script hash, so that a single output can do both. The idea is you make a key, J, and J is an actual key. Like, you have the private key for it. It's a regular old pubkey. And you have a script z. And instead of sending either to the script z, in a pay to script hash construction, or to hash this key J inside the pubkey hash, you say, I'm going to compute key C-- which is J plus the hash of my script, concatenated with J, times G.

So this is the same as the thing I just did with the OP_RETURN and 0 bytes, right? And then I send money to it. And then, what do I do? I've sent money to this C point. Can I sign with C's private key? I know I know a little j. That's a regular key. Do I know a little c?

Sure, yeah. I know that little c is just the j private key plus this hash. So if I want to treat it as a pay to pubkey hash, I just sign with little c. I sign my transaction and I'm good. Nobody has to know that I did this. However, if I want to make it into pay to script hash, I reveal the script z and the subkey J. And then I put some arguments on, and then run the script.

So if I don't want to show that there was a script, I don't have to. I just sign with this. Private key's that. Nobody knows that there was a script at all. So if everyone's using this, 80% of the

time that's what happens. Maybe there was a script in there. Maybe there wasn't. Can't tell, right?

However, if I want to reveal, hey, actually this time, I'm using the script. So I revealed J and z. Everyone computes-- OK, well, if I add J to the hash of z and J times g, does that equal c-- the key that is in the output script itself? And if it does, cool, that was a valid proof that you actually committed to z. And now I run the script z. Any questions about this? Make sense?

So it's actually pretty simple. It follows directly from the proofs, from the contract hash. And it's really cool because-- oh, P. I thought I was doing C. Oops. Anyway.

So what you can do is you can make the sum of everyone's keys. And so in Schnoor signatures we can do this, and we can sign without revealing each other's private keys. So you can say, OK, what I'm going to do is J is the sum of, like, 50 different pubkeys. And then z is that Merkle tree of 2 of 50 signatures.

So if I want to use 2 of 50, I have to reveal, and then I have to do the proof. But if I can get all 50 of 50 people to sign, they can produce a signature with private key little C. So when everyone's cooperating, you don't even see that there was a script involved. And you can do this Schnoor signatures. You cannot do this with ECDSA. So probably this would be put into Bitcoin alongside a Schnoor signature kind of construction.

This seems useful because in most cases, most smart contracts, most multisig does have sort of an all-participants-sign clause. Even if we don't code it in right now, it's generally the case. Like, 2 of 3 multisig-- well, 3 of 3 is good. And if you do 3 of 3, hey, you don't even see that there was 2 of 3. And so that preserves privacy. It saves space. And most of the time all the participants can be online. So that's the same basic idea as in lightning. If everyone's co-operative, you don't even see that there was a lightning channel. You just together broadcast a transaction with just 2 of 2 multisig. But in lightning, you do see that there was 2 of 2 multisig.

So you can sort of see, oh, it was either a channel or a multisig. But in this kind of construction, you wouldn't even show that. You'd say, OK, this is actually 2 of 2 multisig, but we can merge them.

Oh, other weird trick-- you can make a public key, and prove that there is no known private key. So what if I only want this script z, and there is no set of everyone signing? I don't want that to be a requirement. What if I only want the script hash? I can do that. Interactively, I can

use someone else's J. If I'm sending you a script that involves someone else, I say, hey, can you come up with a key J? I'm going to compute a key C, that's J plus the hash of the script comma J times G. And then I can prove that to you after the fact. And you can say, OK, well, I know it's my key. So interactive, I can show that it's someone else's key.

Non-interactive, I can just say, look, I know the pre-image of the x-coordinate of J. So basically, I can prove to you that there's no way I came up with this point on the curve in the normal multiply by G way. Because that sort of shows that I came up with the x-coordinate by just hashing a random number-- or a non-random number, if it's the hash of 0, or if it's the hash of anything. The idea is I randomly picked an x-coordinate on this curve, figured out what the y-coordinate was, turned that into a point, and then used it in this construction. If I do that, you can be pretty sure that I don't know the private key for J, because I didn't compute it the right way. Does that make sense?

So you can do that. That's a way to then prove to other people like there is no J. It's a point on the curve, but it's a random point on the curve. So there's no way I can sign with c. I sent the pubkey c. And I proved to you, look, C is not a key that anyone can sign with. However, there is a point J, and there's a script z, and you can execute the script. So that's useful in many cases, where you want to prove to people that, look, there is no key here. It's just a script. And then everyone will find out later when you spend it. But they don't have to know that there was no key. They just know that the script got executed. So that's a nice sort of patch, where you can default. If there's any use cases where in the current script hash you want that exact same functionality and you don't want this new functionality, you can sort of get rid of it this way.

AUDIENCE: So the idea of [INAUDIBLE] is that you are trying to make your special transaction scripts not stand out [INAUDIBLE]. The fear is that people will see those and not put them into blocks, or not [INAUDIBLE] them, or discriminate against them?

TADGE DRYJA: Maybe a little. But it's more that-- let's say I have my wallet software, and it always uses to P2WSH, P2SH, or something. Then I can say, oh, I bet this is the same person. I can try to track who is spending them.

AUDIENCE: [INAUDIBLE].

TADGE DRYJA: Yeah, a big part of it is privacy. Maybe they're discriminated against them like miners. Probably, the miners don't care. But it's more like if everyone's using software that can create similar transactions, then it becomes much harder to tell people apart, and it has more privacy

for the transactions.

AUDIENCE: Then this one, you're putting in a script that can anybody can potentially execute it, but they don't know that they can?

TADGE DRYJA: They need to know both the script and J, which is not obvious from C. C is the only thing you see on the blockchain, in the output. And so they just said, it's a key-- probably, someone knows the private key. Maybe there's some weird thing like this going on. You don't have to use this, right?

And then, yeah, so they won't know what J or z are.

AUDIENCE: Still you're [INAUDIBLE].

TADGE DRYJA: Yeah. You can publicly publish it, if you want-- and tell everyone in the world, hey, this C is actually this J plus this script z. They still can't spend it. That can be public, even beforehand. And then they see, oh, there's a script and a key. And you can also, say, sign something, and prove that you know the private key for J. So you can say, I can prove that I'm not doing this-- there is no J kind of thing.

So you can prove all those things, but you don't need to. For normal usage, you just send a key C, spend from key C. You're done. And if you want to spend from the script z, you just provide J, the script , and then execute the script. So it's nice. And you might have had a script, and then you didn't use it, so you didn't reveal that there was a script.

So it helps with privacy. It helps with efficiency in many cases, because if you don't need to use a script, you don't have to. So in a lot of cases you say, all three of the 2 of 3 multisigs are on board, so let's use C instead of the script z. So it's pretty cool.

OK, so questions about this? Go to the next crazy thing. Little notes about taproot-- anyone can make a key and script and send to it. Only the pubkeys are needed. So if I want to send to something that's 3 of 3-- Alice, Bob, and Carol added all together-- or 2 of the 3 of Alice, and Bob, and Carol-- I don't have to even ask Alice, Bob, and Carol. If I know they're public keys, I can just compute all this myself, and send money to them.

And I can compute any construction I want. I can make it 1 of 3. I know Alice, Bob, and Carol's public keys. And I can say, hey Alice, Bob, and Carol, I'm giving you all money, and you can fight over it-- or you can cooperate. Like, any of you get it. And so the idea is 1 of 3 multisig

script goes into here, and then 3 of 3 is here. So if you cooperate, you save a little bit of space, and the fees are smaller.

But if you don't cooperate, any of you can grab it immediately. I don't know how useful that is, but you could do this. You don't need knowledge of any private keys to construct C. Only the pubkeys are needed. That's actually really nice, because you're creating. So the same is currently the case in Bitcoin. If you're creating an output script, like pay to pubkey hash, pay to script hash, there's no private keys involved in that process. You just tell everyone.

OK, which differs, and is the important distinction between the next cool thing, which is graftroot-- also Greg Maxwell, like a week or two after taproot. They're not actually that similar. But it was like, wait-- people were sort of talking about it.

So the idea is allow a lot of scripts with a proof that goes up in size with 0 of 1-- aka it doesn't go up in size. So the Merkle proofs-- the MAST-- go in log n. So if you've got a million different scripts, you're going to take something like 20 hashes or whatever to prove that. What's a proof that grows in 0 of 1 of the number of things being proven? It's actually not-- it's kind of obvious.

Like, I want to prove something, and the proof size does not change regardless of how many things I am proving. Well, it's just a signature. If I make a million signatures, they're all the same size. So I could, instead of making a Merkle route, I just have all these little leaves. And I just sign them all. And a leaf with a signature is just as good.

So the idea is this is called graftroot. It's similar to taproot in that it's got sort of two execution paths. You send to a key C, and there's two ways to spend from it. So you can use the regular pay to pubkey hash mode, where you're just spending from C. You just sign with key C, your transaction-- same as in taproot, same as in regular Bitcoin right now. That's the easy one.

The script version is I show a script s, and a signature from key C on the message s, and then I execute that script. So this is kind of cool, right? You say, look, I'm sending you a key, and either that key itself signs or that key endorses a script to be executed. And I can make as many of these as I want. So if I want 1,000 different scripts, it's not like I have to prove inclusion in the three-- that's going to be height 22, or height 12, or whatever-- I just have a signature, saying, OK, key C signed this to a 50 multisig. Key C also signed the other two of 50 multisig.

So the one that I'm actually using, I just provide the proof. The proof of inclusion is just a signature. Questions about this? Makes sense? Kind of weird?

It works. The real problem is there is this key C, right? So there's this sort of root key that controls the whole thing. The idea is key C endorses all the different leaves. So in the case of MAST you've got, let's say, four different scripts that can execute. All four of those are committed to by the Merkle root.

In the case of graftroot, let's say you have four different scripts. C signs all four of those different scripts. The thing is, C can sign 10 scripts. C has got total control over this output. Whereas before, in the MAST idea, it can be different people. It could be Alice and Bob, Carol and Dave, Edna and Gertrude-- I don't know. And they might not be friends. No single entity there has full control.

So the idea of this is, well, you can combine a whole bunch of people. The key C is essentially everyone, and they all have to sign off on all the different scripts that could happen, and then distribute those signatures on those scripts, which are proofs.

OK, basic idea make sense? This one's a little weirder. In this one there is a way to prove that there is no key C. So that's to sort of say, look, it's only going to be this script, and no one's going to sign in the normal just the key signs method.

There's also a way to do that here. It's much less useful, though, because what you essentially do is you start with your script, and then you compute a signature on it-- but you also prove that there was no private key that made the signature. You have time. It's actually really fun.

What you do is-- normally you say, OK, S equals k minus the hash of my message and R times point a. What if I want to prove to you that there is no a? SG equals R minus the hash of message R times pubkey A. I want to prove that I do not know that private key, and nobody knows it.

Actually, I have to both. I say, OK, I know the pre-image of the x-coordinate. Same with R. Wait, can I do this? I forget how this is going to work.

Anyway, basically what I do is I compute A public key randomly. Or no, I first get the message I want to sign. I make a random R, and I show that it's random-- I know the pre-image of its coordinates, or something. I can prove that I didn't construct this. I don't know the private key.

And then I solve for A and S, essentially.

So I can show, look, I've got a signature, but I can prove that no one signed. I think I make a random s, and then I solve for big A. I make a random s, multiply by G, solve for A.

So A divided by hash of m, R-- wait, I can't do that, can I? Yeah, I can. This is [INAUDIBLE.] Equals R minus SG. I come up with a random R, put it in here, come up with the message I want to sign, and then solve for A. So I can do that, and I can prove that there was no real A, because of the way I constructed it. I know weird things about R and A. And so I can show that this is not a real signature.

However, in this case, it's not that useful. In the case of taproot, it's useful, because I might want to prove to people, look, this is just a script. There is no key. In this case, I can still do that. I can say, there's just a script. There is no key. But then I can only do it once. The whole idea of graftroot is I can have a million different scripts, that are all going into one key.

OK, so the problem is the root key must sign every script. And now you need to use private keys to create an address. That's not great, right? It's maybe OK, but you haven't had to do that before. So now we have to sign off on every script that we're endorsing, in order to compute an address.

You can do this after the fact, though. So that's kind of the interesting thing about this, is you can say, OK, I'm sending the key C-- which is this root key, that's all 50 participants. And then after the fact-- after it's gone been confirmed in the blockchain-- those 50 participants can say, actually, you know what, 2 of 50 is good. We're not suspending the money right now. But we're all agreeing that, yeah, any two of us can spend this money later.

And so they all get together, and all 50 sign with the root key C all the different scripts-- those 1,225 scripts-- and then hand those signatures out to the respective people who would be interested in that signature.

So, oh, Alice and Bob, if you want to spend the money, we all made a signature endorsing your 2 of 2, 2 of 50-- your two keys. And Bob and Carol, if you two want to spend the money, we've made a script that's endorsed by us saying, your two keys can spend it.

It's really cool. The overhead is one signature. So you can have that like 100 trillion different-- 50 choose 25, which is 100 trillion. Well, you've got to sign 100 trillion times, but when you spend from it, it's small. It's an extra 33 bytes of overhead. Yeah, the overhead is just a

signature. So you say, OK, here's key C. Here's a signature from it. Here's the script, to endorse the script being executed. It's not actually 64 bytes. You can get the overhead into 33 bytes, because you can aggregate the S values of the signature.

I'll talk about this on Wednesday. I don't think I actually came up with this, but Greg cites me on it, and it's like, I guess no one thought of this. But there's a non-interactive partial aggregation of these things, which I was really hoping would get into Bitcoin itself, but I think it's too slow. But in this case it's not that slow, so people are like, yeah, we'll use it here.

So this is kind of cool, because it's simple. It's just more scripts can be added any time. It's all one scaling. So if you have it a million different scripts, proving one of them takes 32 bytes, or 33, or something. C can be a threshold signature of many different parties. The signature can be aggregated within the transaction. The downside is this interactive setup, where you can't just take Alice, and Bob, and Carol's key, and send to a graftroot 2 of 3 multisig. Alice, and Bob, and Carol need to sign something.

In some cases that's no big deal. In some cases, you're connected to them over the internet, and all the participants are cooperating. Sure, we'll sign. We'll create addresses. Or we'll pre-sign all sorts of different addresses beforehand. Doable, but sometimes you don't want this interactive setup. Sometimes I want to be able to send to participants of my choosing, without their cooperation. Really, usually for sending people money, they're OK with it, right? But sometimes you don't want them to be online.

So put this all together. The goal for this, and I think the way it's going to work in Bitcoin-- but this is still research-- you have a unified output script. So let's say it's a templated with OP_5. And you say, OK, OP_5, pubkey.

Now, when I want to spend to it, I have a bunch of options. To spend, I can just put a signature on the stack. And then the program says, oh, there's a signature. Let me check if the signature matches the public key-- you know, this pubkey. And if it does, cool, the money moves. That's pay to pubkey hash mode, except it's not hashed.

Another thing I could do is provide a point J-- a script-- and a series of arguments for the script. And that's taproot. I verify that J plus the hash of J in the script times G is equal to that pubkey, and then I execute that script.

Another way to do it is if I've got this-- I've got C, which is also just a point and a signature on

the script-- and then a script, and a bunch of arguments, then I say, oh, this is graftroot. I'll verify this signature on this script, and then execute with these arguments.

So it would be cool if you could do it just based on the number of items pushed. But you might want to put like a flag or something here, to distinguish between these two. And with this, if there's just one element, you can just quickly say, OK, it's pay to pubkey.

But to distinguish between these two there might be weird edge cases, where it's like, well, one of the arguments could also look like a script, and annoying things. So you might put some, like, flag byte But anyway, it's kind of nice, in that when you send to an output, you send to an address, all the addresses look the same. They just look like public keys, but they could be any of these things, and you only find out after the fact.

And not only that-- you could have the same key have a taproot and also have a graftroot. So there could be an address. And I've gotten my wallet. Well, there's three different ways I can spend this.

I can spend this with everyone-- all 50 people-- signing. Or I can spend it with a single script that's been endorsed this way, that's got priority. And then I can also spend it with one of the millions of scripts that have been endorsed this way. So I can use all three. This is a little bit more efficient, because you're just revealing the script and a point. This one, you have to reveal.

Oh no, wait, sorry, this is wrong. There's no C. You already know C. It's still a little more efficient. Yeah, there's no C there. You already know C. You just put the signature on the script. Yeah, OK, so that's [INAUDIBLE]

OK, so this is graftroot, taproot. And then, also, part of the script can be masked. You can have a masked OP code. You can combine all these things in crazy ways, and have like really powerful scripting that way.

So when this stuff was proposed, the argument was, hey, I don't want to spend to a public key. I want to spend to a public key hash. Because quantum computers are coming. And when the quantum computers come, if you know my pubkey, you can just calculate my private key, and take all my money. But if you only know my public key hash, quantum computer doesn't help you there. You're not going to be able to compute the pre-image of a hash with a quantum computer.

All of those statements are true, except in practice it's dumb. Because if there's a quantum computer that can compute your private key from your public key, the only time using pubkey hashes will help you is if somehow that quantum computer takes a really long time to compute your private key. Because if it only takes them a minute to compute it, well, you have to reveal your public key to spend your money.

And then the attacker, who has this nifty quantum computer, can just say, oh, that's his public key. I just saw it. And he signed. And using his public key, I just compute his private key, sign a different message, and broadcast it. And maybe I attach a higher fee. Because I don't care, it's not my money. And the miners are like, well, there's two different transactions-- one sending over here, one sending over here. This one has a much higher fee. I'll use this one. And the attacker with the quantum computer still takes all your money.

They can't take it at rest. They have to wait for you to try to spend it and take it. But you could say, OK, well actually, I'm doing OP_5, pubkey hash. And then for this, I have to reveal pubkey signature. For this, I reveal j component of pubkey script. For this, I reveal C and the sig. But it's kind of ugly. It wastes space, and in most cases doesn't provide a lot of protection. So there's hopefully better ways to deal with this. One thing that I have sort of been joking about, but would be kind of fun to put it into [INAUDIBLE]. You can actually use taproot to commit to a Lamport-like hash-based pubkey.

So in taproot this is actually quantum secure. I hesitate to advertise this. But it is a commitment that the quantum computer cannot forge. Where was I saying the J equals hash of-- yeah, this thing is still a problem for a quantum computer. I want to compute a J such that J equals this hash times G minus R. And J is in there as well. Even with a quantum computer, maybe I can compute an arbitrary J and find its discrete lock-- you know, find the private key-- but it's still not going to satisfy this equation.

So I can't after the fact find a script and a J to compute this C. So that is safe, even if a quantum computer exists, because that's based on the security the hash function.

So that's kind of an interesting little tidbit-- like, hey, this taproot construction is hash based. Quantum computer appears, and we say, look, we no longer allow C. We no longer allow the normal spend, where you sign with key C, because signatures-- all this elliptic curve stuff, it's broken.

However, we do allow you to prove that there was a script z committed to in C. And then maybe this script has like a quantum-safe signature in it, and then we can use that. What's nice about that is you can start using it sooner, and without the overhead of all the extra space taken by the quantum signature, because it's sort of a backup plan. It's like, well, normally I'm just signing with C, because I know this is just my regular key-- and I do this. But worst-case scenario, if a quantum computer appears in 10 years, I've got my other backup key here. So that's kind of a cool thing. I don't know how useful it is in practice.

So yeah, you've got all these three things. The other thing about this is-- I don't know, big multisig is one use case. It seems really powerful, and like, hey, I can make these millions of different scripts, and Merkle trees of different scripts, and taproot and graftroot. Not a ton of really good real world use cases, though, I think that would motivate it a lot more.

Because a lot of the hesitation from people working on Bitcoin is like, OK, this is really cool, but what are we going to do with it? As I'll talk about on Wednesday, multisig itself is not the best motivation. Because once we have Schnoor signatures and aggregate signatures there's direct ways to do multisig, that are actually more privacy preserving and more space preserving.

So we need to use cases. And that's one of the debates-- like, OK, what do we do with this? This is kind of cool. You can make all sorts of weird smart contracts, but what are some real-world use cases? There's also sort of argument-- OK, MAST versus graftroot versus both. Well, if you have graftroot, do you really need MAST anymore? Instead of this big Merkle tree, you just have the signatures endorsing the leaves. So it sort of seems better scalability there. But graftroot is interactive, so maybe you still want MAST. Maybe you have both. There's all sorts of debate on this.

And so in practice that may mean that since people are arguing next year, the year after, it's hard to get everyone on board to do stuff. There is code, though. Some things have code. A lot of it's sort of obvious enough, that like, yeah, graftroot makes sense. Just have them sign. The taproot construction there's code for. But it's not all put together. There's no spec for all these things.

So if people are interested in this stuff, start coding it. Start trying it out. Come up with cool use cases, and maybe push for one thing or another to get in. There's a full request open in Bitcoin right now for MAST. I know some people are really into it, and other people are like, uh,

I don't know. Also, how to implement MAST exactly.

So this is still research kind of level, but it seems that they're usable, and it may improve smart-contract functionality. So cool stuff here. Any questions about this?

All right, cool. So I'll do office hours tomorrow at 4:00. We can talk about projects and stuff. And the next class on Wednesday will be even more crazy elliptic curve stuff about aggregate signatures, Schnoor signatures, multi signatures, threshold signatures, things like that. And also, privacy-- I'm going to focus on the signature part. I also put privacy and CoinJoin in there, which is related, but might be squished towards the end if I don't have a ton of time.