TADGE DRYJA: Today, I'm going to talk about signatures and all sorts of different signature kind of things. In the problem set, you're working with signatures, but you're working with hash-based signatures, which are not actually used in bitcoin at all. But we'll talk about those.

OK. So if you've gotten through the homework, there's lamport signatures, right? These are hash-based signatures. And they use hash functions. So it's fairly straightforward. You can understand them. There's nothing super crazy going on. The code is fairly compact. So that's cool.

What are some disadvantages of these lamport signatures? Does anyone-- yeah?

AUDIENCE: You can only use it once.

TADGE DRYJA: Yeah. OK. So plus. This is hashes. That's cool. One-time use. Other possible disadvantages of them relative to other systems, if you're aware. Another is they're kind of huge, kind of big, right? You can deal with it.

But if you were looking in the forge, or that file with the signatures, it's like, what, 8K for a signature-- 8 kilobytes-- kind of big. Keys are 16 kilobytes-- kind of annoying. Private keys are also 16 kilobytes. So yes, sig 8K, 16K priv/pub key. So that's some disadvantages.

So since I don't have slides, I'm gonna make this more fun and interactive. What are some solutions for these problems? So we can actually mitigate/solve both of these things to a pretty good extent.

So how about the first one, one-time use? What would be a fairly obvious way to mitigate the one-time use problem? And don't think the answer is too stupid. It may be a fairly stupid answer, and it might work. So yeah?

AUDIENCE: Not actually revealing pieces of your private key? Instead, reveal something else.

TADGE DRYJA: There's probably some clever way. But that might be too clever. Something really simple for, OK, I can only use a key once. How can I use a "key," quote unquote, more than once? Yeah?

AUDIENCE: Make another one.

TADGE DRYJA: Yeah. You can make another key. So you could say, well, I've got this 16 kilobyte public key. Well, I'm going to make a 32 kilobyte public key. And it's just two public keys stuck together.

And now, when I make a signature, I just put an extra bit in the front. And I say, well, this signature is using key 0 or this signature is using key 1, and it's got the whole signature after. And then you look through this 32 kilobyte public key, and you say, OK, well, it starts with a zero, so that means it's using the first key, the first subkey in this 32 byte public key block. And in this case, it's using one, so that means it's using the latter subkey.

So that would work. That would let you use your public key twice, at the cost of doubling your public key size, which is not really great, right? And it's not very efficient. But it does sort of work. OK. Any clever ways to do it more efficiently? Or wait. So OK, also, I'll give you sort of a hint.

In this case, let's say this is pub sub 0 and pub sub 1, right? And then, your 32 byte pubkey is just them concatenated together, right-- pub0, pub1. What would happen to the private keys in this case, right? How would private keys work here? Same expansion of size, I guess.

Can anyone think of a way to mitigate the expansion of size of private keys in this case? So the private keys are the preimages here, right? They lead into these public key blocks. So you could just say, OK, well, I have twice the size private key leading into twice the size public key. Could you do that more efficiently? Yeah?

AUDIENCE: Could you just hash the private key so that you have two hashes instead of one?

TADGE DRYJA: Yes. So let's say you have this 16k block, and you want this to turn into two public key. So that's the basic good way to do it. And it sort of turns in like that. How exactly-- what's the way you do that?

AUDIENCE: You can keep the same private key as before and just add something like zero or one to indicate--

TADGE DRYJA: Yeah. So this is a hash function, right? And so before, we just said, OK, hash of this is block 0,

this is block 1, this is block 2. So the idea of pub-- let's see. Is this visible? This might be too small, right?

AUDIENCE: Yes.

TADGE DRYJA: Yes, OK. Let me make this bigger. Sorry. OK. So in these diagrams, you've got your private key right now. And it's in these big blocks. And there's 256 of them, but let's keep it small. And the idea is these are 32 byte blocks with random numbers in it. And then you hash it to get your public key.

So we say, OK, pub2-- and this is public, this is private, and, let's say, secret-- pub2 is just the hash of secret2, right? But yeah. What we could do is we could sort of have two different hash functions.

And then a real simple way to make a whole bunch of different hash functions is we define, OK, well hash0 is defined as the hash function of whatever your input x is concatenated with the number 0. And then hash1, we define as just x comma 1 and so on. And this is actually secure. You could do this. Any questions or possible objections?

AUDIENCE: I was thinking that if someone knew the hash function you're using, wouldn't they only define x because they know that it won't help with [INAUDIBLE] x. [INAUDIBLE].

TADGE DRYJA: Yes. Yes. So there's no real entropy or secrets in this 0 and 1. But it's purely riding on x, right? But the idea is, well, if I do this, and I say, OK, well pub2 is the hash of secret2 concatenated with 0, yeah, if you know secret2, you can go back. Because 0 is obvious.

But the idea is if you don't know secret2, the fact that you know the last byte of the hash input doesn't really help you. Because there's all this data that you don't know. And so you're not going to be able to find a preimage. You're like, OK, I know the preimage to public2 ends with a 0 byte. What are the other 32 bytes that come before that? You still can't go back to make a preimage.

AUDIENCE: But it feels like there's some sort of-- what's the word-- you'd make a similar statement saying, oh, if the last byte is not important, then the second-to-last byte is not important, either, right?

TADGE DRYJA: So the attack is I've got pub2, which is a 32 byte hash output. I want to find what this thing is. Now, I do know what the last byte is, right? It's going to be, let's say, 33 bytes, and the last byte is a 0. And this is going to be some random numbers all the way back.

So I have some insight into what the preimage is. I know the last byte is a 0. But that doesn't help because I don't know all the bytes before that. So I can't really do-- I can't sign because I need to be able to reveal all these things.

And the way the hash functions, if it's a good hash function, the fact that you know some part of the preimage shouldn't give you insight into the other parts of the preimage. Because it looks really random. And adding stuff to the end here shouldn't reveal anything.

There are hash functions where this can be dangerous, and you can consider those like broken hash functions, where if I concatenate a lot of known data at the end, it might give me properties of the public key that I can find. [INAUDIBLE], as far as we know, it works fine. Most of the hash functions work fine this way.

So yeah, you can do this. And then you can say, OK, well, I'm going to use hash function 0 to make pubkey 0, right? This is pubkey 0. And if I use hash function 1, I make pubkey 1.

Now I have a 16 kilobyte-- let me erase this-- so now what I can do is I can say, OK, well, I have a 16 kilobyte secret key, private key, but I have a 32 kilobyte public key that I can use twice that has two subkeys so I don't have to store as much of my private key data. So that's cool. That makes it quite a bit more efficient.

Does anyone have an idea of how to get it even more efficient for private key storage? You can actually get it very efficient using this kind of idea. Any ideas? Yeah. Well, you--

So some hints. This is pretty useful, right? The idea is, well, even if you know what the sort of suffix of all these things are, if you don't know x, you're not going to be able to find the preimage, right? So maybe add some more stuff here. Any idea? Oh, yeah?

AUDIENCE: Could you hash the index?

TADGE DRYJA: Right. Right. So instead of just saying 0 and 1, we can say here, well, why not just have one chunk? And there's one chunk, and it's got random number x. Well, this is hash x comma 0. This is hash x comma 1. This is hash x comma 2. OK. So yeah.

So what you can do is you can-- this was before, right, where I said, OK, yeah, but this is basically how to do a 32 byte private key where your private key is actually quite small, and you just derive all of the public keys by adding numbers at the end, concatenating them into your hash.

-

And now you're like, OK, cool. I have a 32 byte private key, and I can make enormous public keys from them, and it's still secure, right? Because I can add whatever number I want here. And I can do this a million times, and no one will be able to find the preimage, even though I'm giving you all these related hashes, right?

It's x comma 0, x comma 1, x comma a million. I can give you millions of them. And each time I do it, it doesn't give you any insight into how to find what x is-- if it's a good hash function.

So that's really powerful. And now I can say, OK, the private key problem is solved. I have 0 of 1 growth in my private key. I can make as many public keys as I want, never gets any bigger. Cool.

Still have this problem with the public keys, though. Let's say I want to sign four times. OK. Well, I make a 64 kilobyte pubkey. Now I can sign four times. Great. But my public key size expands with the number of signatures I ever want to do.

Any ideas on how you could mitigate that or how you could still make a usable system, despite this fact? There's a couple different techniques. Any ideas? OK. So one would be every time I sign, I also sign my next pubkey.

You can do that. It's got some downsides, right? So the idea is I first publish one 16 kilobyte pubkey. And then I say, OK, when I sign, I'm going to sign my message. And concatenated to that message will be my next pubkey. And so I'm signing something and also indicating what my next public key will be that I sign with. Then I can still maintain a small private key and then keep signing.

The problem is, in order for someone to verify, they're going to need to look at all of my previous signatures, right? So if I'm saying, hey, here's my 500th public key that I'm signing with, here's the message, you're going to have to go back and look at my entire history of signing, which is not great.

OK. So other ideas of how to deal with large pubkeys? What's a way you could do this? So hints. Is there any kind of data function we've seen here where it takes an arbitrary sized amount of data and outputs a fixed size?

[LAUGHING]

In fact, it's the only thing we've been talking about so far. So what would you do to make your public keys smaller?

AUDIENCE: Hash.

TADGE DRYJA: Hash them. Right. OK. So you say, well, yeah, 32 byte pubkey. Well, you just commit to the pubkey with the hash of the whole thing, right? And I say, well, I've got my 16 kilobyte by pubkey. I just hash it, right? And now I have a 32 byte public key. And I've committed. That works, right? I've committed to my public key.

However, when I sign, I need to provide the whole public key in order for you to verify. I need to provide this whole 16 original pubkey, original version pubkey, after when I sign. So now my signature goes from 8K to, what, 24K. So I haven't really made it more efficient, right, in that the total thing with public keys and signatures stays the same size-- actually, slightly larger.

But this is kind of a game. Because the public key, it might be like I'm showing it to you beforehand, and we're space-constrained there. And then, when I'm doing the signature, I have more space or time or something like that. So this is actually useful.

And actually, this technique is used in Bitcoin and all different cryptocurrencies, where you can call this a pubkey hash-- pubkey hash, or they say PKH. And the main benefit is size, right? You could say, well, if I've got a really big pubkey, I can commit to it by sending to the hash of the pubkey. And then, when I later want to sign, I reveal the actual pubkey. You can do it that way.

OK. So then this is pretty cool, right? We've now taken our system, which had fairly large-- 16 kilobyte-- private keys, turned them into 32 byte private keys. Great. And also, this is 0 of 1-- never expands. Awesome. We've taken a system with large, potentially huge-- 32 kilobyte, 64 kilobyte-- public keys. We've got it down to 32 bytes. Awesome. OK. We've now made this thing much more efficient, much more useful.

The problem-- the signatures are still big. They actually got a little bit bigger because they include the full public key in the signature. And let's say we had one of these things, where we make a bunch of public keys and commit to them at the outset.

So you say, OK, I've got a 64 kilobyte pubkey. I can commit to the whole thing by hashing it. However, then, when I reveal and want to give you a signature, my signature is now whatever it'll be-- 72, right? The signature would be 72 kilobytes-- kind of huge.

So I can add a bunch of keys, but I'm just sort of pushing the problem somewhere else-- into the signature, right? There's a better way to commit to many public keys. It's maybe not super obvious unless you've seen it before.

But if you have, do you know what the better way to commit to many public keys is? Or you could come up with it on the spot, and it'd be like, wow, you should have gotten this stuff named after you. Any idea? OK.

So what you can do-- it's a little non-obvious-- you can make a tree, right? So the idea is instead of just saying, oh, I'm going to take all four of my big pubkeys, stick them all in a row, take the hash of that whole thing and publish that as my public hash, I'm going to have a little bit more complex design. And that design is a binary tree.

OK. So the idea is here are my four public keys that take up 64K each. I can make these intermediate hashes. So I make a binary tree where I say, OK, this is the hash of the first two keys stuck together. This is the hash of the second two keys stuck together. The top is the hash of these two intermediate hashes stuck together.

And then, this top is my root, which is equivalent to my pubkey hash. And so I can publish this, and it commits me to all four of my public keys. So it might not be obvious. Do you see any advantages that this could have over just concatenating all four of the keys and hashing it? Yes?

AUDIENCE: I'm confused. Where is your private key in this?

TADGE DRYJA: The private key is down here, right? The private key is not shown. But I have a private key. It turns into these pubkeys. And we can use the technique here, where we just have, really, one 32 byte private key and create all the different chunks of all the public keys by concatenating different indices. Yes?

AUDIENCE: You can do [INAUDIBLE] systems and log-in time instead of linear time.

TADGE DRYJA: Yeah, exactly. I can prove that, hey, this is public key 0, which is in here, this root hash, and I don't have to reveal all four in order to do it. So yeah. Did someone else have, probably, the same idea? Yeah, OK. OK. So how are we going to do that? You obviously know. Anyone else? Yeah.

AUDIENCE: So assuming you wanted the public key 0, you would publish that and 1, and also hash of 2, 3.

TADGE DRYJA: Right, yeah. And then you can prove up to the top. So I have little colors. So the idea is the verifier who's verifying the signature, he already knows the root, right? Because you've pre-published that as your public key, as your public key root and your public key hash.

And then that verifier is provided with public key 0 saying, hey, here's a signature. Here's the message. Here is public key 0. So you've got both of these things. Now, the prover needs to prove that public key 0 is contained within the root.

And to do that, the only two things the prover needs to add is pub1-- OK, so the prover says, here's public key 1. That allows the verifier to compute this. And then the prover says, hey, here's hash 2, 3, which allows the verifier to compute the root and then check that it's equal.

So yeah, you only need two extra hashes instead of all four. So saves some space. It doesn't save that much space when you only have four. It saves a lot of space when you have 1,000, right? So you can add 0 of n elements. And the root stays the same size. The root stays 32 bytes, regardless of how many leaf nodes you have.

And you can prove an element with log n-- oh, those parentheses are wrong-- 0 log n intermediate hashes. So if you have 1,000 keys, your root stays 32 bytes, and your proofs are going to be not too, too big, right-- 320 bytes of overhead, which is a bunch smaller than a signature, right?

So this is really cool. And you can use this much more practically than the raw lamport signatures that were in the first problem set. Because those, if you're just doing one key and one signature, it's about the same size. But this actually lets you use it multiple times, lets you store a much smaller private key, commit to much smaller public keys. It's a more usable system.

The signatures, though, still are kind of huge, right? Your signatures are going to be a little over 8 kilobytes, which is OK, right? Computers can deal with 8 kilobytes. That's called a Merkle tree made by Ralph Merkle in '76 or something. And it was originally made for exactly that purpose.

Anyway, so these are called Merkle trees. Ralph Merkle. He doesn't really work on crypto now. He works on biotech stuff now, I think. Anyway. But he's cool. And he gets, like, everything named after him. So you will see the word Merkle everywhere.

And it's, you know, he invented something cool. Awesome. This is one of the kind of things, though, where it seems like, yeah, I might have been able to think of that, right? It's sort of a binary tree and hash functions. It's nothing that crazy.

But so there's all sorts of Merkle trees. You can make a different thing. It's used in Bitcoin. It's used in a bunch of different cryptocurrencies. Oh, cool. It's a very powerful way to take a whole bunch of different things and commit to them in one small thing.

OK. So this is very cool. But we can do better. And so the next things are things that I definitely would never have thought of because it's much more complex than just, like, hey, let's use a binary tree with hashes.

And I will mention RSA and then go into ECDSA and elliptic curve [INAUDIBLE] signatures, which do different things and are not based on just hash functions. So I'll talk about RSA for a few minutes, and then we'll have a little break, like intermission. And then we'll go into elliptic curve stuff.

OK. So RSA was invented by locals. The R is Ron-- no, I think the R stands for Rivest, but Ron Rivest is at CSAIL, and he's still working on cool crypto stuff. It's not used in Bitcoin or any currency, actually-- well, any currently operating cryptocurrency. Because the signatures and keys are a bit larger than the later systems we'll see-- smaller than the hash-based things, but still a bit larger.

Was used in chaumian blinded cash. Because you can do blinded signatures, where you say, OK, I'm going to have you sign something for me, and then after you give me the signature, I can pull apart the signature.

Because I'll have you sign something that you don't know what you're signing, right? I'm going to mask this message and blind it so that you sign the message, and then I can get your signature on the real message without you knowing what you signed.

So basically, the idea is, well, I actually have message prime equals message plus randomness. And I say, hey, can you sign m prime? And you say, OK. I produce a signature on m prime with my private key. And we'll call that s prime.

And then what I can do is I can s equals s prime minus the randomness, or, really, some function of that randomness. And now I have a signature. And this is a valid signature on m.

So that's kind of cool. It's like, hey, can you sign this message for me? And we're not going to tell you what you're signing.

It seems sort of useless, like why would I sign something I have no idea what I'm signing? But it can be used in that chaumian blinded cash. Because I then have a proof that, well, this bank didn't know what they were signing, but they signed this, and this is their signature, and they gave it to me. And I can use that as a thing to represent money. There's all sorts of cool things you can do with this.

The basic setup of RSA is make two prime numbers, p and q. And then compute n, which is the product p times q. So this seems fairly straightforward, right? It's not too hard. Everyone knows what prime numbers are. Now you have this composite number, p times q.

How do I make a leading question for the next slide? Does this have any relation to things that we've looked at? This doesn't look at all like a hash function, right? But are there any properties similar to a hash function here? Yeah?

AUDIENCE: It's hard to route the image. It's hard to [INAUDIBLE].

TADGE DRYJA: Yeah. It's a one-way function in that if I know p and q, I can compute n, right? But given n, I can't necessarily find p and q, right? If they're small, sure, I just check. And there's all sorts of algorithms to find factors.

But if p and q are pretty big, like hundreds of bits long or thousands of bits long, finding n may be impractical, whereas if p and q are really long, finding the product of both of them will be very practical. Even if huge, multiplying two integers is really easy.

So this is a one-way function but not a hash function. And so a lot of these signature systems use one-way functions, but trickier one-way functions that aren't hash functions, and still have cool things that you can do. So I'm not going to go into RSA in super detail. I'm going to run through it really quick, and you'll be like, huh, how does that work?

But if you think it's interesting, you can look it up and then maybe you go to CSAIL and ask Ron how it works, although maybe try to look it up first. He's probably explained it a lot of times.

But it has different properties. So you could say, I'm going to use this p times q equals n as a hash function and make a lamport signature out of it where my private key is a whole bunch of

p and q pairs, and my public key is a whole bunch of n's, and I reveal them and stuff. You could do that. That would work.

But you can do much more powerful things because of the mathematical properties of these things. So yeah, I'm just going to go through it really fast. But if you know anything, this is kind of cool.

The idea is you can do a bunch of multiplication and exponential modulo n. And so because n is almost prime, there's some interesting properties, where if you know the factors of n, you have these shortcuts you can do. And someone who doesn't know the factors of n can't do these shortcuts.

So you publish e, which is a constant-- this is sort of a parameter system. So RSA is used today all over the place. If your computer is open and you've been using the web in the last five seconds, your computer has performed RSA calculations. Every time you connect to a website, you're doing RSA.

So generally, it's 65537. They just make this up. You could also use three-- some small number. And then d is a number that you can compute if you know what p and q are. So this is sort of the private key, right? p and q itself can be seen as the private key. But you can actually compute this d number and then discard p and q.

And so this is how you compute d or whatever. But the idea is n is your public key, and d is your private key. e is always the same, and p and q, you can discard afterwards. And then what you can do-- and not going to get into it-- but you can say, OK, my signature s is message, right-- I can take the hash and make a message m-- message to the d power modulo n.

And if I verify, if I take that signature to the e power modulo n, it should equal m again, which is crazy and kind of awesome. Because the idea is to sign, OK, I just raised to the d power mod n. It's going to be some number on the same size as n, right?

And then I give that to someone. They take that signature to the 65537 power modulo m again and see if m-- the message-- comes back out. And if it does, they know I signed, and I have the private key that corresponds to n.

Now, the cool thing is I can use this any number of times. I can use the same thing a bazillion

times with different m's, and that doesn't give any help on how to factor n or how to compute d. So this was late 70s, also, RSA, and one of the first, hey, we can do better than hash functions.

It was developed at around the same time as hash-based signatures. All of these things were sort of coming out at the same time. You can do lots of cool stuff because there's these properties where we can make m prime plus m and take the exponentiation of all that, and it still works.

OK. So that's really cool. You can do lots of cool stuff. But this is also not used in Bitcoin. One of the things is the size. So key sizes are smaller than with lamport signatures. And they're often about 2 kilobits, or 256 bytes, which is fine. And that's the same size as signatures.

So if you use your web browser-- secure-- well, how secure? Certificate, valid. Oh, this browser might not show me. OK, never mind.

AUDIENCE: In Firefox, it'll do it.

TADGE DRYJA: In Firefox, it tells you more. OK. Well, anyway, anyway, anyway. But yeah, there's RSA keys in, basically, every certificate for web browsers and websites. And they're generally 2,048 bits long. That's seen as pretty secure. Even 1,024 bits has not been hacked. But people think it probably could be by a very motivated actor. So this is decent size.

One thing about RSA is that it's a little tricky to implement, right? There's a whole bunch of weird math there. It's not like the lamport signatures, where it's like, OK, I have a hash function, use it, right? It's kind of hard to shoot yourself in the foot with the lamport signature scheme.

You can, right, but it's sort of obvious if you do, so if you're doing the homework and it just doesn't work at all, and you're like, oh, I was hashing twice, or, oh, I was reversing the order, or it just didn't work. But there's no subtle ways where, oh, I was off by one, and now it reveals all my private keys.

That's hard to do, whereas in RSA, it's actually quite easy. If your p and q values, for example-- what's a really crazy gotcha-- if p or q minus 1-- so if p minus 1 or q minus 1-- is smooth-- essentially, a number that's highly composite has many small factors-- you can find someone's private key. You can factor it.

So it's not just that p and q have to be prime. P minus 1 and q minus 1 have to be not too unprime. It can't be a power of two or something. Otherwise, you can lose your privacy. So there's all sorts of crazy things that happen with RSA.

Bitcoin and other coins use elliptic curve signatures, which I'll talk about next, that are sort of displacing RSA, in many cases. So it hasn't really taken off for web certificates but a lot of other types of encryption, signing, things like that.

Cryptocurrencies are also one of the big uses of elliptic curve signatures because they're somewhat more powerful and somewhat smaller and really cool. And we will talk about those in three minutes. So intermission, three minutes. Stretch. Ask random questions. Talk to the neighbors.

OK. So they're using these elliptic curves, which are curves of the form y squared plus x cubed plus some coefficient x squared plus some constant. And in the case of Bitcoin, it's very simple. It's y squared equals x cubed plus 7. And that's the curve Bitcoin uses. And this is what it looks like if you plot it. And that seems really simple. We can totally understand that. But it gets a little weird.

With elliptic curves, you have these points on the curves, right? And another property is if you draw a straight line that intersects this curve-- you could also draw a line which doesn't intersect at all, fine-- but if you draw a line that intersects the curve, it will intersect in three places.

So if I draw a line here, it intersects these three dots. If I draw a line here, it intersects here, here, and then it'll intersect somewhere down here. It'll intersect in three places unless I take a tangent. If I draw, and I try to be like, oh, I'm going only intersect in two, well, I can take a tangent here, and it'll intersect in one more place.

What we can do, though, is we can say, well, if you intersect at a tangent, that's kind of like intersecting twice. We can deal with that.

AUDIENCE: What if you draw a vertical line?

TADGE DRYJA: If you draw a vertical-- oh, yeah, vertical line, then there's a point at infinity, which shouldn't happen.

AUDIENCE: But infinity is included--

TADGE DRYJA: Yeah.

AUDIENCE: --in the set of points on the curve? So it's like there's three points, anyway.

TADGE DRYJA: Yeah. There's a point up there-- sort of. In the practice of signing and stuff, you probably shouldn't hit it. Yeah. But so what we do is we say, OK, well, three points in a line equals zero. And so if we take this point plus this point plus this point, that's zero.

Another way to say that is P plus Q minus R equals 0, or P plus Q equals negative R. And another property is the negation of a point is just that same point with the y-coordinate flipped. So if this is negative R, we just go down to the bottom side of this, and this is positive R. Similarly, minus P would be here. Minus Q would be here. And if you had a point Z here, negative Z would be up here.

This is how it's constructed. However, we don't actually use these nice curves. We chop them up in modulo something. Yeah?

AUDIENCE: You have P plus Q equals negative R. Should it be positive R?

TADGE DRYJA: No. In this case, R is up here. Right. In this case-- P. Wait, hold on.

[INTERPOSING VOICES]

TADGE DRYJA: Oh, shoot. Sorry. P plus Q equals R. Right. Sorry. OK. Never mind. Yeah. The picture is correct. But so basically, P plus Q equals R because R is down here. Sorry. Yeah, that's why. OK. This is R. This is negative R. So yeah, P plus Q equals R, not negative R. Sorry.

When you want to multiply a point, you can. Because you can take a tangent, right? So you can add these points, right? You can say, OK, I want the sum of P and Q, which will be down here-- R. I want 2P, for example. Well, with 2P, I can take a tangent and then find where it intersects and then go down below the curve.

So for example, if this is G, I say OK, well, G starts here, take the tangent, find where it intersects and then drop down. And this is 2G. So it's a simple way to say, well, I'm not really multiplying. I'm just adding it to itself, which is sort of what multiplication is.

And I can do this again and again. So I say, oh, I want 4G. I take the tangent at 2G, find the intersection here, drop down. OK. Here's 4G. Take the tangent here, it'll go-- I can keep doing

that and get 2G, 4G, 8G. And if I want intermediate things, like, well, I want 3G, well, I draw a line between G and 2G, find the intersection, and find the sum. So yeah? Yes?

AUDIENCE: When you find the intersection of that [INAUDIBLE] the intersection of the modulo of something, does it just--

TADGE DRYJA: Yeah, yeah. So the pictures look nice, in this case, right? It's like, hey, there's this curve. And it looks so simple. And we can graphically do it on graph paper. But when you're actually doing it on the computer, it looks more like this. Everything is modulo some giant prime number.

You could still draw lines. It's just that the curve is now all these crazy-looking dots. And you still have that symmetry from the y-axis. But the way the computer does it is all modulo a number. So it doesn't actually map to a nice curve.

So the fun thing about when you're doing modulo is all the properties stay the same, right? With regular numbers, you can still do addition, multiplication, division all modulo some number, and it still all works, right? So same with this. You still have the same draw a line between P and Q, and you get to here. And then you flip over the y-axis, and now this is R.

Similarly, you can take a tangent-- although a tangent is harder to visually see because there's no obvious curve here-- but the equations work the same. OK. So you don't need to know exactly why these things work or curves. But we're going to go through what you can do with these and what properties you have.

OK. So we've defined these points on a curve and how this curve works. And then we also have regular old numbers, right? So what we'll do from now on is say, like a and b, our lowercase numbers, these are scalars.

Scalars just means regular old numbers. The uppercase A and B will refer to points on the curve, OK? So from now on, lowercase is regular number, uppercase is point on this curve, or point on this mashed-up thing that we still call a curve.

OK. So I'll go through what operations we can do, and then we can see what use those have. So first, scalars. You can add, subtract, multiply, divide. You can do whatever you want, right? These are just regular numbers, like five or 17 or 22. Everything's OK, right? Easy.

Everything here is integers. We're doing modulo some big prime number, so there's no decimal points. I think in this class, we'll probably never have to use floating point, which is

great, because floating point always scares me.

It's also uint. These are like unsigned integers. You can do it with signed. But since you're modulo some big number, it's like unsigned numbers that loop around. So it's really nice for computers. Computers deal very well with these things.

Computers are good at floating point, as well. But the thing is, in cryptography, a lot of times floating point ends up with error, right, where you're pretty close, but you've got some fixed amount of decimal places you can store. Your floating point goes out to some precision. And then error can accumulate.

And in a lot of cryptography, you want zero error, right? Like in a hash function, it's pretty exact. So a lot of times, the cryptography, you use integers instead of floating point or rational numbers.

OK. So with scalars, these are just regular integers, natural numbers even. You can do these. All the operations, you're familiar with. That's fairly straightforward.

With points. So you can add and subtract the points, right? We showed how you do that visually. And then the equations for it, the computer can do that pretty quick. However, this is not defined. Multiplication and division with two points is undefined. It's just not clear how you do it. It's not something in this system, right? It's a group. You've got one operation. You don't have this other one.

So you can add and subtract points, but you can't multiply them. Any questions about this? This is pretty important. Make sense? Yes?

AUDIENCE: In the previous case, when you were adding points, what is defined as addition? Are you adding only the y components?

TADGE DRYJA: You use both the x and y-coordinates to add, right? So when you're doing it in the computer, you're saying, OK, well, what are the x- and y-coordinates of P? What are the x- and y-coordinates of Q? Find the slope here and where it intersects the axis. And then see where it'll intersect the curve at another point.

AUDIENCE: Addition Is literally defined as compute that point. It is not a function of adding the x- and y-coordinates.

AUDIENCE: Gotcha.

TADGE DRYJA: Yeah. Well, I mean-- yeah. But the equation would be-- since the computer doesn't actually look at the curve, you just say, OK. Find the Px- and y-coordinate. Find the Qx- and y-coordinate. Find the slope, and then find where it intersects, and then compute for that for R.

OK. So yeah. You can add and subtract. You cannot multiply and divide these points. Sounds good? Any other questions? Yes?

AUDIENCE: How would you subtract?

TADGE DRYJA: Ah. You just go down, right? So if I want P minus Q, I should say P plus negative Q. And negative Q is just right here, right down on the axis. So I say, OK, I want P minus Q. I go here. Find it. It's going to be up there somewhere. So yeah. You just add the negative of the number.

I guess that's another thing. Minus B-- negative B-- is also an operation I can do. I can negate something, which lets me do subtraction.

AUDIENCE: For those of you with math backgrounds, this is defining a group operation on the field of natural numbers, modulo--

TADGE DRYJA: Which is a big one.

AUDIENCE: --and using this elliptic curve. So this is A. And the only thing that's defined is addition and subtraction and not multiplication.

TADGE DRYJA: Right. You can do the same group operations without using elliptic curves. And so you can just do exponential modulo big prime number. I'm going to use curves because it's what is used in all the actual systems. But you can sort of forget about the curve after this, right?

You can say, well, look, we just have these uppercase variables and these lowercase variables. And the fact that it's on a curve, we can sort of abstract away and just know that we have these different types, essentially. And when you're in the computer, you're just like, yeah, that's a point. That's a scalar. Here are the operations I can do, and I just run the functions.

OK. So the next part, when you mix scalars and points, so mixed operations. You cannot add a point and an integer or subtract a point and an integer. That's not defined, right? You say, OK, here's this point p minus 7. Well, does that mean move the x-axis? Does that mean move

the y? It's not defined.

However, you can do this, right? You can multiply and divide. So you can say, well, A times 2, well, you take the tangent, right? It's just A plus A. Or A times 7, is we just break it down as A plus A plus A plus A 17 times.

We can also divide, which is a little weird because we have to find the inverse of B and then multiply by that-- the multiplicative inverse of B modulo the order. But anyway, you can do these.

This is a little trickier because you have to compute what, like, B to the negative 1 is and multiply by that. You can do it. So you can do these things where you have points times scalars. But you cannot add. Yeah?

AUDIENCE: So 2A is basically joining the tangent to A, [INAUDIBLE] beginning.

TADGE DRYJA: Yep. So that was here. If this is G, this is 2G. Find the tangent, find the intersection, and then negate. OK. So you can do that. That's cool. You cannot do that. OK.

So roster of operations we can do. This is it. And this is sort of, if you think of that, you can do all these cool things. It's sort of amazing, all the crazy things you can do with just these operations, right?

With regular numbers, you can do whatever you want. That's obvious. And with these points, you can add and subtract them. And when you mix, you can multiply and divide. And that's it.

OK. Any questions so far? You don't have to worry too much about the curves and all of the math and stuff. But you just sort of say, OK, here's my toolbox. Here's what I can do. OK. Any questions? Good?

So now what we can do is we can define a one-way function sort of like a hash function, sort of like multiplying p times q to get n. What we will need, in addition to this, is everyone pick some point on the curve, and we call it G. And it's random. And we want it to be somewhat verifiably random. We just all have to agree on a point. And we'll call this the generator point.

Another property is that, since it's all modulo, this number, if B is too big, you'll wrap around, right, because it's all modulo, this n. So it's called a generator point because-- and every point on the curve can be one-- if you keep, say, 2G, 3G, 4G, 5G, eventually, you'll get back to G,

right? It's all finite.

OK so does anyone have an idea, OK, how would we make some cool one-way function given these operations and the fact that we have an agreed-upon point? It's not super obvious, but it's actually fairly straightforward. Once you see it, you're like, oh. Any ideas? Yeah.

AUDIENCE: So you take your point G, and you multiply it by some scalar, and you publish where the point is [INAUDIBLE] scalars.

TADGE DRYJA: Yep, yep. OK. So your private key-- let's say private key, lowercase a, it's just some 256 bit random number, scalar, exactly the same as 32 bytes like in a lamport signature. And your public key is just a times G. And your public key is now a point on a curve. We're going to call it uppercase A.

And you have a 32 byte x-coordinate, right-- because the x dimension is 256 bits-- 32 byte y-coordinate. So it's 64 bytes-- pretty small. Anyone have another quick optimization on how you could reduce this? So there's a lot of optimization going on in all these different cryptocurrency systems. Because they don't scale too well. But any idea of how to get that down?

AUDIENCE: Can you hash it?

TADGE DRYJA: You could. So you could hash it. You'd make it into 32 bytes. There's a more useful way. But if you hash it, you still have to reveal it later, right? So it helps, making a pubkey hash. And that's what Bitcoin did initially is you'd have the 64 byte public keys. And then you'd send to the hash of them.

There's a nicer way. Any ideas? OK. So the thing is it's symmetric, right? So the curve is symmetric about the x-axis. Any ideas? yes.

AUDIENCE: Could you just code one of the x- or y-coordinates and say which side?

TADGE DRYJA: Right. So you take the x-coordinate and encode that. And then just that one bit-- 4-- it's on the top. It's on the bottom. And then let them figure out exactly where it is, right? So you can encode the x-coordinate only and 1 bit for y, and then you're down to 33 bytes.

It's a little annoying because the 33 bytes, well, one of those bytes is going to be empty. It's just going to have one bit in it for up or down. But whatever. So you can do that, and that's also really cool. OK. Any questions so far?

AUDIENCE: Could you encode a [INAUDIBLE] and same up and down? Could you just encode an [INAUDIBLE]?

TADGE DRYJA: Huh. Probably. But I don't think it would be smaller. I think you'd have to probably have the same size if you use some other coordinate system. Hm. Yeah, you could encode a slope--

AUDIENCE: Yeah, the [INAUDIBLE].

TADGE DRYJA: And you say it intersects, and it's at the slope, and which side of the axis?

AUDIENCE: Yeah, and if it crosses multiple times, which one of the--

TADGE DRYJA: You probably could. I think you would end up the same size. But yeah. But it might be faster for some things. There's all sorts of different encodings.

AUDIENCE: I guarantee that you have an integer in the end.

TADGE DRYJA: Yeah. So you might not be on the curve. But there's a similar problem here where it might not be on the curve. You can encode a point that's also off the curve.

And so when you actually get a public key, like over the wire in these systems, the first thing you usually do is make sure this is a valid public key, and it's actually on the curve. Because there's a lot of points that aren't. And sometimes you can screw around with the code that way. Yeah?

AUDIENCE: Can you define the scalar multiplication again? how does it go?

TADGE DRYJA: OK. So scalar multiplication-- a times G is, basically, break it out as G plus G plus G plus G plus G a times. And G plus G is find the tangent of G and then find the intersection, right? So this is G. This is 2G, right-- tangent, intersect, and then the negation. And so you just do that a bazillion times.

However, since you can find 2G and then you can double 2G to get 4G and double 4G to get 8G, you can come up with powers of 2 of G and then add those powers of 2 to do a much more efficient addition so that you can practically make-- because little a is going to be some huge decimal 50-digit number or whatever. So you're going to have to do this two, four, six, eight binary expansion and then add them all up. OK. Yeah?

AUDIENCE: You're chucking everything into some grain size to get these. You're [INAUDIBLE] a

continuous curve.

TADGE DRYJA: Yes.

AUDIENCE: It feels like if you have 306 bits in the original, you shouldn't be able to get all the way down to 32 bits, right? Because you just want to make sure that you--

TADGE DRYJA: This?

AUDIENCE: Yeah. If you removed a byte of precision, would you actually be losing any power?

TADGE DRYJA: What you can do a little-- what you can do is you can just say, look, I'm only going to allow public keys that are on the top, positive y component. And everything else is disallowed. You could have that as a rule of your system, and then you can get rid of the y bit.

But the thing is the y-coordinate does affect some of the things like signatures, things that you'll later use. So you can just say, OK, look, implicitly y bit is positive. And you lose 1 bit of security. Because there is a difference between the positive and negative thing if you're doing signing.

But yeah. And so some systems do do that. And it makes things easier. There's some signing systems that do that. OK. I'm going to go to the next part.

ECDSA is used in bitcoin. I'm actually not going to explain it. It's a worse signature system. So the reason why this was used was the better system was patented. And so there's a clearer, more obvious, cleaner, more powerful signature system that was patented by this German guy, Claus Schnorr.

And so no one used it, or maybe a few people used it. But in general, with open source things and web standards, patents are really hard to work with. And so all the cryptographers said, well, we'll make this other system that's different enough that it doesn't infringe on the patent. And it's kind of ugly.

It does work as a signature scheme. But actually, I'm going to explain the Schnorr signatures, which make a lot more sense, are easier to understand and manipulate, and the patent has expired, so we can all use it. And it will probably be going into Bitcoin. And some cryptocurrencies use Schnorr signatures.

I think Monero uses a Schnorr-like signature. But Bitcoin will probably be putting it into Bitcoin

in the next year or so. So all these things will be much more applicable, and you can use them. And there's code out. You can use them now.

OK. Yeah. Sorry. Patent has expired-- free to use the better algo that must not be named. So yeah. People don't call it Schnorr signature as much. Because it's like, well, this is the guy who prevented us from using this for 20 years. And there's modifications to it.

OK. So the elliptic curve signature, for lack of a better name. This is something of a simplification. And there's reading about how you can really do it the right way. But this does give you the right idea. And for a single signer, this is secure, and it works. But later things, you might want to actually add other stuff.

So the idea is you've got your message m, right? Same as in lamport, same as in RSA, you've got a message m and a private key lowercase a. And your public key is a times G, like we said. And then, when you want to sign, you actually make a one-time use key pair for the signature only. And they usually call that k.

And k is a new random number. It's the private key. And you're going to multiply it by G. And they call this R. I don't know why they use these letters, but anyway. K times G is R. R is the other public key you're using just for this signature.

And then the signature itself is quite straightforward. As the signer, you compute s, which is k-- this new private key you just made up-- minus the hash of your message concatenated with R, this public key-- so you encode the public key in 33 bytes, stick it in there-- multiplied by a, your normal private key. And then your signature is the pair-- the R point and the s scalar. This is a little bit of a bunch to process. But it's actually not too bad, right?

This is a scalar-- 32 bytes-- that you made up. The hash, this is a message, also going to be however long. This is 33 bytes. You concatenate them together-- hash. You end up with another 32 byte scalar. You multiply those two scalars. So actually, the only elliptic curve operation here is calculating what R is. When you're actually calculating s, this is all just scalar, so it's very fast.

So in terms of practical computer stuff, doing these operations is a little bit on the slow side. Because you're doing those point additions quite a number of times. So a decent code, a decent CPU, can do on the order of thousands of these types of operations per second, which is decent.

You can do a decent amount, but it is kind of slow. And it's considerably slower than hash functions. You can do a million iterations of a hash function per second on a core, or you can do, maybe, 1,000 or 2,000 of this kind of thing, so still pretty fast but slower.

And so this signing procedure, this takes a bit of time. This takes, actually, very little time. Because you're just subtracting and multiplying 32 byte scalar integers, which is much faster. OK. So does this make sense? We'll go a little bit of how you can't forge this.

OK. Sorry. So first verification. Sorry. So how would you, if anyone knows or can think, how would you verify this, right? I know their public key-- capital A, which is little a times G-- and then they're giving me a message m, and they're giving me a signature, which is R and s. And they've computed it this way.

I can't verify this equation myself because I don't know what k is. I don't know what little a is, right? I'm just given R, s, m, and big A. So how would I verify this?

AUDIENCE: Can I, for example, take the hash of the message, use R, and then try random a's and that-- on, sorry.

TADGE DRYJA: Yeah. Do you have one?

AUDIENCE: So trying random is probably [INAUDIBLE] because--

TADGE DRYJA: There's a lot, yeah.

AUDIENCE: --there's too many. The old one that you don't know it. But I think one more thing you could do is take little s and multiply it by big G.

TADGE DRYJA: Yes.

AUDIENCE: And then do some--

TADGE DRYJA: Yeah. So the basic idea-- yeah, you're right. Multiply this side by G. Then you also have to multiply the other side by G, right? So you say, OK, well, as the verifier-- I've got R, I've got s-- I multiply s by G, and now I have to multiply this side by G. And what'll that look like? Well, what's k times G?

AUDIENCE: R.

TADGE DRYJA: R. What's little a times G? Big A. So yeah. I want to verify. I want to verify that this equation was used, right-- s equals k minus this hash times little a. Multiply both sides by G, I get s times G equals k times G minus the hash thing times little a times G. This is big A. This is R.

Hey, now I know everything here as the verifier, right? I know what s time G is. I know what R is. I know what big A is. I know what's going into this hash function.

And so I just say, OK. Well, I rearrange it a little, and I say, is R equal to this? And if it is, that's a good signature. If it's not, it's not a good signature. Any questions about this? Does this makes sense, mostly?

So for the verification, it's a bit more CPU-intense, right? You're given a scalar. You multiply by G. You also have to multiply A-- this is actually more costly-- you have to multiply A by this hash. The reason that's more costly-- it's still a scalar times a point.

The thing is, since G is used for multiplying a lot, you precompute all sorts of coefficients times G, and you store them in RAM or in your CPU somewhere. So multiplying by G is a little bit faster because you do it a lot, you can precompute a lot, whereas multiplying by A is different each time for all the different signatures.

So this is a little bit slower, in general. So it takes a little bit more CPU time to verify a signature than to create one, something like twice as much. But still, you can do this pretty fast. On a modern computer, you can do thousands of these a second. OK. Any questions so far? I'll go a little bit into why this is not forgible, right?

So I want to forge a signature. Well, I don't know little a. But why don't I make up a k and compute s and R? But I need a, right? So I'm going to say, OK, this is the equation I want. I'll just make up my own k and make up my own k, s, that'll satisfy this.

Without a, I really can't make a valid s, right? I can make something up here. But I can't compute it because I don't know here, right? I can try to grind through hash functions, but it won't work. The basic problem is if I make up an s and solve for R, right-- I can say, well, I know what R is-- but the thing is R is in the hash function, as well. It's the one-wayness of the hash function that actually breaks this, right?

So I could say, well, let me solve for R. Well, R is the hash of m and R times a plus s times G. If I can come up with a valid R here, I can do this.

The problem is R is defined by the hash of R in this case. I can't compute this, and I can't cancel this out, right? So you can see just like, OK, I need to solve for R. Well, but it's already in here. I'm stuck. If I don't know k, I can't come up with a signature, either.

OK. There is one footgun with EC signatures. If you use the same R value for different signatures with the same pubkey, you reveal your private key. So k has to be random and new every time. If you look through the equation, you'll see how you can solve-- if you have two of the same k's with different s's-- you can solve for k.

And then, once you solve for k given this, if you know what k is, you can find what little a is, right, as a verifier. You can say, well, you gave me s. That's part of the signature. I know this whole coefficient. If I know k, I can solve for little a by dividing this out.

So always use a new k value. There have been many cases where people don't. Probably the most famous one is PlayStation 3 used ECDSA. And they used a fixed k value for signing all their code. And that allowed people to calculate their private keys and then run pirated games.

That was actually the first time I learned about elliptic curve signatures, in 2010 or something. So actually, there's interesting tutorials written by the hackers who broke Playstation 3 about how this works. And that's when I first read about it.

OK. We're almost out of time. Any questions about this stuff so far? OK. So this is a signature algorithm. It's like, OK, fine. This works. I can sign, same as lamport signatures, same as RSA.

What's really nice about the elliptic curve stuff is you can do more than just signatures. There's all sorts of fun things you can do. And it seems like a fairly unexplored area in that I've found fun things that you can do that I guess are novel. And I'm not an expert on this at all. It's just that it's a fairly new area of research in how to do cool bitcoiny cryptocurrency things with these curves and points and stuff. So I'll go through a couple fun things you can do.

Fun with points. OK. So I'm not using an asterisk. Just b times G. So let's say Alice has a public key A. Bob has a public key B, right, which is little a times G or little b times G. Well, this is weird. Little a times big B-- Alice's private key multiplied by Bob's public key-- you can do that, right?-- is equal to Bob's private key times Alice's public key, right? It's commutative. The multiplication is commutative.

That's weird, right? It's just a times G times b, which is the same-- or it doesn't matter the order-- it's the same as b times G times a, which is the same as a times b times G. And let's

call that C. C-- this is called a Diffie-Hellman key exchange point.

Diffie and Hellman are two also late-'70s PhD students who came up with all this cool stuff. No, wait. Diffie was a PhD student. Hellman was his advisor? I don't remember. Anyway, they came up with this idea.

And this is a shared point. What would this be useful for? It's actually super useful. Can you think of like, oh, OK. We can do this. We can compute C. Why would we want to do that? Any applications you can think of? Yeah?

AUDIENCE: Trying to share keys with people without both of them knowing who they are and evaluating the other set of keys.

TADGE DRYJA: Yeah. It's really cool to share keys. So this is also done if you have a web browser. I think it's all elliptic curve Diffie-Hellman now, mostly. But if you have a web browser and you go to a website, this is happening, basically, for every HTTPS request. Because you can use C as a sort of shared key and encrypt things with it.

The idea is big A can be public. Big B can be public. Point C will not be public. The only way to compute it is to know either Alice's private key-- little a-- or Bob's private key, little b. Even though you can see the public keys, you can't compute C from just the public keys, right? Because big A times big B, that's not defined.

You can't multiply the points. You have to multiply the point by a scalar. And since you don't know the scalars from viewing this exchange, you can't compute C. But Alice and Bob can both compute C. So this is really useful.

You can compute C. You can use it for encryption. You can use it to prove you are who you say you are. You can use it as an interactive verification where you say, OK, I'm Bob. I'm going to make random point. I'm going to give that to you. Prove you know your private key-- little a-- by telling me what C is, right?

So I can make a random key here, give it to you. You return C. I verify that that's the right C. And I say, OK, well you must know little a. You haven't shown me a little a, but you've proven that you know it, so proof of knowledge of the private key.

So you can use that for logins instead of passwords. And it's much simpler than a signature. So that's really cool. All sorts of things you can do with that.

The last part, you can have fun with points. And I use this in my software. So you say, OK, well, there's a public key A. There's a public key B. And we'll define D as the sum of these two public keys.

The thing is if you're summing these two public keys, it's commutative, and it is the same as saying the sum of the two private keys times G, right? a times G plus b times G is the same as a plus b times G, which is another really cool property.

So what if I compute this public key D, and I say I want a signature from public key D? You can. And you so the private key, little d, which can sign for this, is just little a plus little b. So you can make a combined key. And then either party-- Alice has little a, Bob has a little b-- and they could reveal it to each other and allow the other person to make a signature with D later.

I use this in the lightning network software I've had, where, basically, you say, look, we'll compute D, and I can give you the private key, right? I'm Bob. And I say, OK, here's point b. Here's point a. We add them together.

And then, Alice, if I give you my private key, now you can sign, but I can't, right? So I'll give you the private key to D from some information I have. Or you could give me the private key to D by sharing little a with me.

So this is another useful thing where you can have something where both parties know that neither party can sign unless they give them something. And you can use that for bitcoin addresses, things like that. Before you even get to signatures, you can share keys and stuff.

There's all sorts of super fun things you can do with these points and curves and systems. Multisignatures-- I put a link to a PDF file that's very recent and kind of overkill and over my head, as well. But you can do signatures where you aggregate the signatures. But

There's ways to say, OK, well, I'm going to add all these R points, and I'm going to add all these s points. And I can have signatures that are from a bunch of different people that collapse into the same size as a single signature. And I can verify that they're all signing the same thing or, possibly, that they're all signing different things.

So I have 10 different signatures from different people signing different messages. But I can add up all the s values. I have to keep the R values. But now these 10 signatures, instead of being 64 bytes each are more like 32 bites each. And there's one extra 32 byte value for all of

them.

So there's a lot of really cool ways you can combine things, make things smaller, make things really versatile, where we can share keys, things like that, which you can't do with hash-based signatures, hash functions. And so it's really fun.

One worry is that if quantum computers ever become really a thing, all this stuff stops working. All the elliptic curve stuff doesn't work anymore. You can figure out what people's private keys are, whereas hash stuff, hash things like lamport signatures, still work.

So there's some talk of hey, we should prepare for if that ever happens and make efficient, effective hash-based signatures and use them in something like Bitcoin or Ethereum. People haven't yet because the overhead is really high. It's really big. The real thing is all these fun things might disappear in the next 10, 20 years. Who knows? But there's lots of cool stuff.

What do we do with this? So we're not going to do anything yet in terms of problem sets. It's a little hard to program it directly. But there's libraries. And the libraries have fairly easy use, where it's multiply by G is a function, and multiply by this point is a function.

So this is groundwork for cool stuff you can do. And it's really nice because it's a new area. And I'm not an expert on this, but I've come up with stuff that people have cited me on. And they're like, yeah, I guess no one thought of that before. There's a lot of things.

There's one very recent one called Taproot, where it's one line. You're like, oh, yeah. That works. And it's just these sort of elliptic curve operations. It's like, yeah, that works. That's really useful. How did no one think of that? It's one line. Wow. So there's just not a ton of people working on this kind of stuff. And so it's kind of fun.