MITOCW | 1. Signatures, Hashing, Hash Chains, e-cash, and Motivation

NEHA NARULA: OK, so let's get started. OK? So great. We're here to talk about cryptocurrency engineering and design. I think the first question that comes up that's very obvious is what is a cryptocurrency? So this word was kind of invented 10 years ago when-- I don't know how many of you know the origin story of where bitcoin came from, but basically a pseudonym on the internet dropped a paper and some open source code in a forum on an email list, and said, hey, I have this idea for this thing called bitcoin. It's kind of like electronic cash. Here's how I think it could work, and here is some code if you want to run it and become part of this peer-to-peer network.

We don't know who this person is. This person has basically virtually disappeared from the internet and from the world. But it's created something that has captured so many people's imaginations and has sort of, depending on how you measure it, created billions and billions of dollars of economic value and inspired a lot of people to think about how to use this technology to solve a myriad of different problems, not just electronic payments.

So cryptocurrencies and the technology behind them are inspiring people to think about how to bank the unbanked, add more auditability and traceability to our world, get rid of trusted intermediaries and institutions in certain situations, and basically solve every problem, if you read about what blockchains can do on the internet.

Now that's not exactly what this class is about. This class is not going to be about applications. This class is going to be about technology and infrastructure. You're going to learn how to create a cryptocurrency, what goes inside a cryptocurrency, what's important, what are the techniques. And what application you choose to apply that to down the line, that's kind of up to you. But we're not going to be doing digital identity or health care records or something like that. We're going to be talking about the technology.

So a big question is how are cryptocurrencies different from regular currencies? And another thing that I want to make really clear is that the terms in this space are still being defined. So you will hear people throw around all sorts of terms-- cryptocurrency, blockchain, consensus. And these words kind of have floating, evolving meanings right now.

Part of that is because bitcoin, the first cryptocurrency, didn't come from academia, as far as we know. It came from a community of enthusiasts on the internet. And so it doesn't necessarily have the same basis and rigor that we might expect from most of our academic fields of study.

It's totally OK. We're figuring it out as we go along. And academia is really embracing this topic. So if any of you are graduate students who are looking for an area in which to do research, I think basically, the number of papers published on cryptocurrencies and blockchain technology in respected academic venues is doubling every year. So there's huge opportunity here.

So cryptocurrencies are not regular currencies. They're not $1.00 or a pound or a euro, what we normally think of as currency. They're something different. Bitcoin was sort of created out of nowhere. And what does it mean to create a cryptocurrency? Who says you can create a cryptocurrency? What backs a cryptocurrency? Why is it valuable?

Well, first, before we answer that question, I just want to make it really clear what this course is not about, OK? We are not going to help you ICO. If you are interested in ICO'ing, just go. That's not what this class is going to be about. We are not going to offer any trading advice. We have zero opinions on whether you should buy bitcoin now or sell or whatever, or zen cash, or whatever all these things are. So none of that. Don't even ask us. We're not interested.

And this class is not really going to be about permissioned blockchains either. Now you might not know what this term means yet, and that's totally OK, but I just want to make it clear that what we're talking about here are cryptocurrencies. They're open permission with systems in which there is a token which has some value. So that's what we're not going to do in the class.

So going back to-- and let me just pause there for a moment. Let me pause and ask you if there are any questions so far about what I've said.

Yeah.

AUDIENCE: Do they always have to have value?

NEHA NARULA: No, not at all. And let's start to get into that. So the question was do tokens always have to have value? So I think, really, to understand what are cryptocurrencies, what are tokens, what do they mean, we have to talk about money. And we have to talk about what money is and what it means. So this is going to be very hand-wavy and I'm sure not very satisfying to a real monetary economist.

But money developed-- there are a few different theories about how money developed. There is this thing called the coincidence of wants. So maybe I have a sheep and Tadge has some wheat. I am hungry and would like to make bread. Tadge would really like to make a sweater. And so we can barter, we can trade. I have one set of goods that is useful to Tadge. Tadge has another set of goods that are useful to me. We can get together and make an exchange.

So that's fantastic. Barter is incredibly important. Barter has existed for a long time. But what if Tadge doesn't have wheat, Tadge has vegetables, and I don't want vegetables. I want wheat. But Tadge still wants the wool from the sheep. How do we execute this trade? We don't have a coincidence of wants. We don't actually want the exact same thing from each other.

So some theories are that money evolved out of this problem. And money can be represented in so many different ways. Money, I think, was first created around 5000 BC, so it's really, really, really old. The things that represented money usually had certain properties. They were rare. They were not easily reproducible. People, at times, used things like shells or beads for money. The first coins-- this is like a really interesting coin that was developed. Precious metals were often used for money. And then eventually we sort of evolved into what we think of as money now, which is paper bills, currency.

Another theory of how money came about is this idea of receipts, debt and credit. So maybe I have a sheep, and I shear all of my sheep and collect a lot of wool. What I can do is I can store that wool somewhere. And I can get a receipt from someone from having stored that wool, and that receipt is of value. It entitles the person who holds the receipt to the good that is being stored.

And so another theory of money is that money evolved out of these receipts, trading these receipts back and forth. Instead of taking all that wool with you, you leave it in one place in a depository, and the receipt acts as a bearer instrument. Whoever owns it has access to the wool in the depository.

And so you can kind of see two different ideas about what money is develop from this. One is, well, it's a bead or a coin, or something that I hold, something physical that we've decided to assign value to in and of itself. And another idea is I'm going to use a trusted institution. I'm going to deposit something with that institution, and they are going to ensure the validity of that deposit and manage who has access to that deposit.

So this doesn't really get at the question that was originally asked, which is why do tokens have value. But one thing I want to point out is-- well, a question I want to ask you guys, actually, is why do these things have value? Does anyone have any ideas? Yes.

AUDIENCE: Because everyone agrees that they do.

NEHA NARULA: Because everyone agrees that they do. Any other thoughts on why those things have value? Yeah.

AUDIENCE: They're also backed by institutions like the government.

NEHA NARULA: They're backed by institutions. Say a little bit more about that. What does that mean?

AUDIENCE: So the government's kind of promising you to respect the value of that.

NEHA NARULA: OK. The government's promising to respect the value of that. Does anyone want to add to that or have another reason? Yes. And say your name. I'm sorry, yeah.

AUDIENCE: Jared Thompson. In the example of the dollar, the government is willing to accept it as payment for taxes.

NEHA NARULA: Payment for taxes. OK. So that kind of connects the government thing.

AUDIENCE: Even if it had no value of any other sort, it has value in that sense. It's the last thing that holds up its value.

NEHA NARULA: OK, great. Anybody else? Yes.

AUDIENCE: I'm Paul. I think those the three on the front of the dollar, those have inherent value because they might be more rare.

NEHA NARULA: They have value because they're rare. OK. Interesting. All right. So those are all really interesting ideas. I think that those are all sort of properties of what makes things valuable. There are definitely things that are rare that are not valuable, right? I can think of some things that might be extraordinarily rare. There's only one or two of them in the universe, and you would have no interest in owning them whatsoever. You wouldn't assign value to them.

Certainly it's really important that you can pay taxes with this stuff because taxes is pretty much a requirement of living in any country. There are things that have value that you don't necessarily use for taxes. So that's a little confusing. And then there's this idea that it's backed, that it's backed by something. And the dollar used to be backed by something. And actually, if you look at $1.00, I think it still says this, right? It's backed by the full faith and credit of the United States government.

TADGE DRYJA: They don't say that anymore.

NEHA NARULA: They don't say that anymore? They used to say that. But that's what a lot of people say about money. It's backed by the full faith and credit of the United States government. What does that really mean? I think what it all goes back to is these things are valuable because we think they're valuable. We've all decided they're valuable. And you know that if you have a $1.00 bill and you want to buy something from someone, they're going to take it, that you can make that exchange.

And the reason that they're going to take it is because they know that someone else is going to take it. These things hold value because we think that they hold value. It's a collective story that we all tell. So I think once you look at money that way, then when you start to look at tokens, which are essentially digital representations of these things, things that are rare and a little bit special, then when you ask, well, why does this token have value, because we think it has value.

So what makes a token inherently valuable? The fact that we think it's valuable. And a lot of different things can go into that. Maybe we think it's valuable because it's very rare. Or maybe we think it's valuable because someone's promised that you can use it to pay for storage, like with Dropbox. Or maybe we think it's valuable for a completely different reason, because we like the name, or we like the people who are running the network. But ultimately tokens are valuable. These digital representations are valuable because we think they're valuable. Yes.

AUDIENCE: And also because they're a limited amount.

NEHA NARULA: Name.

AUDIENCE: [INAUDIBLE]. Because they're a limited amount.

NEHA NARULA: Well, so my argument is that the fact that they're limited is something that goes into our perception that makes it valuable. Great. OK. So now that we've learned a little bit about money, talked a little bit about money, I want to go into how payments work because ultimately, we're going to get to cryptocurrencies. And cryptocurrencies are electronic cash.

So here's the way that digital payments kind of work right now. You have an institution called a bank. You have Alice and you have Bob, and Alice and Bob have accounts at this bank. And so the bank is keeping track of who owns what. And these are these are records. These might be digital records. They might be paper records, whatever the bank is using to keep track of who has what in their account.

And so the way that I've set up this example right now, Alice and Bob both have bank accounts. Alice has $10.00 with the bank and Bob does not have any money with the bank. So let's say that Alice wants to pay Bob. Let's say that Alice and Bob have gotten together. Maybe they're in the same coffee shop. And Alice wants to buy a sandwich from Bob.

And Bob says, OK, you need to pay me $1.00. If you give me $1.00, then I'll give you the sandwich. So how can Alice do this? How can she transfer $1.00 to Bob? Well, if she had a paper dollar, she could just do that. But let's say that she doesn't have a paper dollar. So Alice can ask the bank to make this transfer for her--or $5.00.

So Alice sends a message to the bank and authenticates with the bank to show the bank that she is, in fact, Alice, but I'm not going to go into the details on how that works. And then the bank confirms that, makes the transfer in its ledger, says Alice now has $5.00 and Bob now has $5.00. Alice tells Bob, hey, I did this. I talked to the bank. Go check. You can verify it for yourself.

Bob checks with the bank and sees, yes, in fact, the bank is saying that he has $5.00 now, whereas before he had zero. And then Bob gives Alice the sandwich because he believes that he now has $5.00. And the bank sort of preserved the property that money was not created out of nowhere, that the balance was ultimately maintained. So the bank is very important in this scenario. The bank is critical.

This is how digital payments work. Credit cards, Venmo, banks, kind of all sort of based on the same idea, that there's some trusted institution that is handling that payment for us and that is keeping track of everything.

Now what are the pros and cons of this scenario? Anyone want to throw a couple out? Yeah.

AUDIENCE: The bank can get hacked and people could move money around between the accounts.

NEHA NARULA: Right. So we're putting a lot of trust in this bank. And maybe should we trust the bank? Banks fail sometimes. Banks are hacked. Banks have humans who are running them who occasionally might want to change those balances in their favor. This has all happened. Anything else? Yeah. And say your name.

AUDIENCE: Brittany. If it's urgent, sometimes you might run into a delay or it might take time with the process.

NEHA NARULA: Yeah. Alice has to talk to the banks, and that's kind of annoying. So there's that. Anything else? Yeah.

AUDIENCE: And if everyone can actually withdraw at the same time, then the bank can actually get money into the system.

NEHA NARULA: So OK. So this is getting a little bit more advanced here. What if everyone takes their balances out at the same time? Well, we need to make sure that the bank actually has that money, so to speak. We're not going to be talking about that problem right now. But very good problem.

So to kind of talk through some of the pros and cons of this situation, one of the big pros, I think, is, that even if Alice and Bob are not in the same physical location, Alice can still pay Bob if they can talk to the bank. So it's pretty cool, and that's something you can't do with dollar bills or with coins or with bars of gold. So having this trusted institution that you can communicate with electronically means that Alice and Bob could be halfway around the world from each other and they can still pay each other. So that's pretty awesome, and that is definitely a property that we want to have.

In terms of cons, I think we covered quite a few of them, which is we're really putting this bank kind of in the middle of everything here. And there are a few different ways that can cause us trouble. So the bank needs to be online during every transaction. If the bank is offline, then how does Bob know whether he got paid or not? The bank could fail at some point in time, which is kind of related to that. The bank could simply decide that they don't want to do this anymore and can block transactions.

And then privacy. The bank has kind of insight into everyone and their payments. And this is incredibly sensitive information. Payments are quite important. And we're going to be talking about privacy a lot in this class, during the second half of this class.

So just an example, a couple of visual examples of that. The bank could just totally go away, and then what happens to that ledger? Who knows, right? I mean, literally, it could just disappear. Maybe it's paper and it gets burnt, or maybe it's bits on a computer and it wasn't replicated. The bank could decide that they don't like Alice for some reason, and that they don't feel like processing Alice's transactions. This happens all the time in the real world.

So there have been designs for electronic cash that work a little bit differently. And we're going to kind of step up to the design that came right before bitcoin, and we're going to do that iteratively. So let's talk about e-cash and how e-cash works.

So the way that e-cash works is Alice tells the bank-- instead of saying, hey, bank, do this transfer for me, Alice says, hey, I would like a digital representation of a coin. Can you give me something that is digital so I don't have to be in the same physical place as you, and that I can use in such a way that I can prove to someone else that I have this thing and that I haven't double spent it, because that's the problem with digital representations of coins.

A fundamental problem is that bits can be copied. So whatever system you use to design your electronic cash, you need to make sure that people can't just copy coins and give what is the same coin to multiple people.

In the previous example, the bank was making sure this happened. The bank was maintaining balances and debiting Alice's account and crediting Bob's account. But if we want to think about something that doesn't involve the bank, and we're starting to get there, then we need to think about how to ensure that a coin can't be what is known as double spent.

So Alice asks the bank for coin. And maybe she has an account with a bank like before. Or maybe she gives the bank teller actual physical money in order to get one of these coins. So the bank generates a unique number-- SN stands for serial number-- and decides that this is the digital representation of the coin. The bank then gives that coin to Alice in a way that it's clear that the bank did this. Usually this is done using a technique called digital signatures. We're going to get to that as class progresses, but not right now.

Once Alice has this coin, then she can give it to Bob. And Bob can take a look at this coin, and hopefully there's enough going on with this coin that Bob can be convinced that this is a real coin. Alice didn't make it up out of nowhere. She actually had the funds, so to speak, to give to Bob, and that it hasn't been double spent. And once Bob is convinced of that, he can give Alice the sandwich.

Now in traditional e-cash, the way that this is done is Bob actually goes back to the bank and says, here's this coin. Alice just gave me this coin. Is this an OK coin? But the fact of the matter is that the bank, in this case, has a serial number and knows that it gave that unique serial number to Alice, and then Bob is showing up with a coin that is that serial number. And what the bank is doing here, in this example, is the way that the bank checks to make sure that this coin is correct is it looks at the serial number, and it makes sure that it hasn't been spent before.

So the bank can link the coin between Alice and Bob, which is unfortunate. The also still sort of has to be online, not to do the actual payment between Alice and Bob, but in order for Bob to have confidence that this coin is real. And later on Bob can say, I would like $1.00 for this coin that I've just given you, or something like that. Or Bob can have an account with the bank and can maintain a balance there.

So just to go through some of the pros and cons here, OK, we've kind of done something where the bank's not in the middle, except the bank is still really in the middle. We're getting a step closer, but we're not there. Alice can technically give Bob this electronic thing that represents value, but Bob still needs to talk to the bank to make sure it's real and it hasn't been double spent.

And we still have this problem where the bank is the one who's minting these things. The bank can decide not to give Alice a coin if it feels like it. And we still have this privacy problem because the secret number, the serial number that we invent for the coin, can be linked across these payments.

So there's this notion of something called Chaumian e-cash. So David Chaumian is a cryptographer, and he developed this system which has slightly nicer properties than previous forms of e-cash. So the idea here, which is really key, is instead of the bank choosing the secret number, Alice chooses the secret number. And we have ways of generating random numbers that we can be fairly sure are unique. So we can let everybody generate their own random numbers.

So in Chaumian e-cash, Alice chooses the secret number that represents a coin. And then

Alice blinds her message. So Alice adds some randomness to the secret number such that the bank doesn't know what that number actually is. And we'll get into more detail about exactly what that means. It's all in the paper that was assigned reading for this class, so make sure that you take a look at it.

So when the bank verifies that the secret number is a real secret number and it's really a coin, and Alice gave the bank $1.00 or something like that, the bank does so on the blinded secret number. And Alice actually has the ability to remove that randomness, or that blinding, later and end up with a valid signature on a secret number.

So Alice does the same thing that she did before. She gives Bob a representation of that electronic coin. And when Bob redeems it, note that the bank never sees what the number is, so when Bob redeems it, the bank has no way of linking the payment between Alice and Bob.

So just to get into how this works visually, Alice will talk to the bank, and Alice will use a blinding factor on the secret number. And so when Alice talked to the bank, the bank doesn't actually see what the secret number is. They can't decode it. Again, Alice gives $1.00 or something like that to get this coin from the bank. And the bank signs this.

Alice can remove the blinding factor later. And this is what the coin is. The coin is a valid bank signature on the secret number, and also the number itself, which Alice can then send to Bob. Bob can check and make sure that this is a valid signature from the bank. And if that's correct, then Bob can give Alice a sandwich.

In order to redeem this, Bob gives this coin to the bank. The bank says, OK, I've never seen the secret number before, and you have my signature on it. So I'm going to assume that I went through this process with somebody and signed something. And now I'm going to record that secret number.

Once that happens, Bob can be sure that this coin hasn't been spent before. The bank keeps a running list of all the secret numbers it's seen, and it makes sure that if it ever sees one again, it can say no, this is not correct. I should never see a secret number more than once.

Now, OK. But know what about Alice could give one version of that to Bob. Alice could also give a version of that to Charlie. And how are Charlie and Bob supposed to know whose coin is correct? Because remember, we wanted to try to get the bank out of the way when doing this.

And so in Chaumian e-cash, the way that this works is the bank actually keeps a bit more information. And the information that the bank is keeping won't let the bank link these transactions together unless Alice happens to give this to two people. And so if Alice gives the same coin to two different people, the bank will be able to detect it and the bank will be able to know it was Alice. And so this is kind of a motivator for Alice not to do that.

So the idea being here is that the way that we get around the fact that we don't know if a coin has been double spent or not is we add punishment if the coin is double spent. So Bob doesn't know for sure that this coin he receives hasn't been double spent, but he does know that if it was, someone's going to know it was Alice, and they're going to punish her.

So this is a pretty clever scheme. And this actually gets us around a lot of problems. We have digital payments. We can make the actual transfer without the bank in the middle. We have some privacy now because the bank can't link transactions together. And we have this way of doing double spend detection. We have a way of motivating people not to double spend their coins, which means that you probably don't have to check at the time you receive a coin whether or not it's been double spent.

Of course, this still suffers from a really big problem, which is that a bank can still decide that they just don't want to do this with you. They can just decide that they don't want to play this game with you. They don't want to issue coins. Maybe they don't like you specifically. Maybe they don't want to take your coins and exchange them. So this scheme, Chaumian e-cash, solves quite a bit of problems when it comes to how do we have electronic money with some nice features, but it doesn't quite get to all of them.

And so the real question in this class is how do we do electronic money, really, in a peer-to-peer way, where there's no institution in the way. There's no sort of entity that can say no.

TADGE DRYJA: So e-cash, the math is really interesting. It kept relying on these banks and so it never quite got off the ground. So I'm willing to talk about somewhat more abstract and low level primitives. I'm not going to quite get into cash or tokens or transfers or anything this lecture.

But I'm going to talk about the really basic primitives that you need that we already sort of mentioned, hash functions and signatures. Signatures, obviously, we talked about a little bit, what you need to be able to sign messages in order to send these tokens around. But first I'll talk about hash functions, which are basically the most fundamental basic thing we use in these systems.

And I think if you've used computers, or if you've programmed a little, you probably have some familiarity with hash functions. They're simple, but they're actually extremely powerful. The hash function is basically you have some data, a bunch of bytes, a bunch of ones and zeros. You run it through a hash function and you get an output that's also a bunch of ones and zeros.

Generally, the input data can be of any size. You can hash something-- put in a megabyte, put in a gigabyte, or put in a single byte, and generally the output is of a fixed size. So in the case of bitcoin, we use Sha-256. The output size is 32 bytes long, or 256 bits long. And this is used for lots of things in computers. I guess the reason they call it a hash is because it's like when you take the potatoes and chop them up into little squares and grill them for breakfast, it's sort of that idea, that we're taking this data. And the data going in gets chopped up and smushed around and then comes out into an output.

So this is not a sufficient different definition. But I will say that you can sort of do everything with hash functions. There's some fun things that you can't do, but you could make a cryptocurrency only using a single hash function. And I think people have, sort of for experimental reasons. You limit the fun stuff you can do, but you can do signatures. You can do encryption. You can do all sorts of things like that.

OK. So this is not a sufficient definition, that there's any size input, a fixed size output, and the output is random-looking. That's sort of wishy-washy. But what does random-looking mean? It's not actually random. If you put in the same input, everyone will get the same output. So if you say, OK, well, what's the hash of one, you'll get some output. And if someone else says, OK, what's the hash of one, you'll get the same thing.

However, the output, while it is deterministic, it's sort of high entropy in that the output should have about as many as one bits as zero bits. If you take the hash of one, it's just going to look like a big random number. And the hash of two will look like a completely unrelated random number. The outputs look like noise. So if you've ever seen hash functions, you can run it on your computer. You say echo. Hello, pipe Sha-256 sum, and you'll just get some kind of crazy, random thing.

There doesn't seem to be any order to the outputs. A little bit more well-defined. We usually talk about the avalanche effect, in that changing a single bit in the input should change about half the bits of the output. So even though you have extremely similar inputs, they should be completely dissimilar outputs-- well, completely dissimilar, as in about half the output changed. If every bit changes, then it just is the inverse of what you had, and so it's easily correlated.

But the avalanche effect is sort of how hash functions are constructed, where generally they're iterative rounds. And so you say, OK, I'm going to swap these things and multiply these things and shift these bits around such that if any change in the beginning will sort of propagate an avalanche, too, so that all the output bits have been affected by it.

OK. And a little bit more well-defined. Generally, the hash functions are defined by what they should not do. So the three main things they should have-- preimage resistance, second preimage resistance, which I'll sort of skip over, and collision resistance. And we can define what these things are.

So a preimage is the thing that came before the output. So it's sort of a math-y term. But the idea is OK, if you know y, you can't find any x such that the hash of x is equal to y. So if I give you a hash output, and that's all I give you, you should not be able to find an input that leads to that output. So if I just say, hey, here's a hash output. It's 35021FF-- whatever, some long string, you won't be able to figure out what I used to put in to get that.

Of course, you can find it eventually. For any given y, there's probably some x. In fact, there's probably a lot of x's that will lead to that y. Since y is a fixed length and there's two to the 256 possible y's, but there's an infinite number of x's because x is not bounded in length. You can have a megabyte or a gigabyte or a terabyte size x.

So since there are sort of infinite numbers of x's, and a fixed, though very large number of y's, as long as it is a random mapping, there will be lots of different x's that can lead to this y. And so you should be able to find it. It's just impractical. It's like, yeah, you may be able to find it, but it's going to take you two to the 256 tries to find any specific y value. And that's about 10 to the 78, which is a number that's big enough that you can sort of round it up to infinity. Well, I mean, not quite, but big enough that you're not going to be able to compute that, the sun'll burnout and the universe'll die and stuff like that.

So that's preimage resistance. You can't go backwards. Given the hash, you can't find what led to that. OK. Any questions about preimage resistance? Seems reasonable? It's a little interesting in that given y is a little tricky, and that it's like, OK, well, someone might know x in order for them to have computed y. Or maybe it's just completely random, and no one actually knows what the x is. So there's a sort of loss of information in the idea of a preimage stack.

OK. Second preimage resistance. This one's a little trickier and can get messy. So I'll define it, but we won't go into it too much. The idea is given x and y such that the hash of x is equal to y, you can't find x prime where x prime is not equal to x. And the hash of x prime is equal to y. So we're sort of giving you a preimage. We're saying, hey, here's this number x and here's this result y. I bet you can't find another x that leads to it.

This one is actually poorly defined in the literature. And so it's a little like, well, who made x, and who gets to choose, and is it any x prime and things like that. So it's not actually that useful. So we can just sort of gloss over that one, just sort of mentioning it.

And then the other one that's very important is collision resistance, where the idea is that nobody can find any x,z pair such that x is not equal to z, but the hash of x is equal to the hash of z. And this one's a lot cleaner in that there's no lack of information. There's no secrets or anything. It's just like, look, no one can find this. And so it's really easy to disprove. You can just say, hey, look, here's an x and here's a z. Try hashing them. Oh, shoot, the hashes are equal. And it doesn't really matter how you got these or who's doing it. So that's a really nice, easy, clear property.

And again, you can find this eventually. So if your output size is 256 bits long, you'll be able to find two inputs that map to the same output. In fact, you do not need to try 256 times. I'm not going to go into the details, but you actually only have to try 128 times. Sorry, two to the 128. So you need to take the square root of the number of attempts in order to find this collision because the intuitive reason is, well, you just start trying things and keeping track of all their hashes.

And there's what's called the birthday attack, which, as you keep trying them, there's more possibilities. The next thing you try, you can collide with any of these things you've tried before. And so you actually only have to do the square root. And it's called the birthday attack because there's the birthday paradox, which is not really a paradox, but the idea is so in this room, there's people that have the same birthday.

It's almost certain, which seems kind of weird because the intuitive thing is, like, well, there's 365 days a year. Maybe once you get 160, 170 people in a room, you're going to have two people with the same birthday. But actually, it's like 22 or something-- anyway, that it becomes likely that people have the same birthday. So it's kind of counterintuitive, and it applies in this case as well. So to find a collision, you need the square root of the output space.

But a hash function should not have collisions. If you can find a collision, if any collision exists for this hash function, you can consider the hash function broken. It's a little bit different than preimage resistance because it's hard to definitively prove that you've broken preimages.

That's something of an interactive process where you say, hey, here's a y, and then someone comes back with an x, and you're like, oh, OK, you prove to me that you can find preimages. But that's hard to tell to the rest of the world because it was sort of interactive, whereas collisions are very clear and non-interactive. You can just say, hey, here's an x and here's a z. Anyone can verify these. Didn't really matter how you got it.

OK. So some practical, how do these functions work. Practically speaking, the collision resistance is a harder property. So there are many functions where the collision resistance has been broken where the preimage resistance has not been broken. So examples are Sha-1 and MD5. MD5's a fairly old one written by Ron Rivest over at-- well, I guess it wasn't at the Stata Center because it was in the '80s. But this was message digest 5. I guess there were several before that. And that is quite broken. You shouldn't use it. Its collision resistance is trivially broken. You can find collisions in under a second on a modern computer.

Sha-1 happened later, in the late '90s, I think, and NSA made it. And there have been collisions found. I think there's really only one collision that's been found, basically, by a team at Google and some Italian university last year. And they spent a lot of computer time to find this collision. But they did find it. And then once you find one, it's sort of like, oh, yeah, we really shouldn't use this anymore. But in both of these cases, sha-1 and MD5, there's no feasible preimage attack. So given a hash output for either of these, you can't find what the input was, or you can't find a different input. So generally, it's a lot easier to make a function strong against preimages. Collisions is sort of harder to deal with.

Also, practically speaking, how do these hash functions work? It's a little bit of black magic. There's no proofs that a hash function can even exist. So if you could prove that there is a one-way function, you get the Fields Medal, right? It's like a million dollar prize. So if you can prove it there is such a thing as a hash function, you will be a super famous mathematician. We have no idea that this is even mathematically possible. Or maybe the universe doesn't work this way.

It seems to, though. It seems like there are these things that work like hash functions, that work like one-way functions, but we have no proof of that. So even the most fundamental part that everything hinges on, we don't even know if it exists. And then this is sort of closely related, if you're in the computer science-y stuff, like p and mp-- anyway, so we don't know that these actually work.

And also, in practice, hash functions are not nice math, cool things like elliptic curves and RSA, prime numbers and stuff like that. They're really, if you look at the code, it's sort of like, well, I'm going to take these bytes and I'm going to swap them. And then I'm going to add these two numbers, and then I'm going to rotate the bits over here, and then I'm going to x over these things. And then I'm going to do that 50 times. And why 50? Well, it seems like 50 is a good number. It's not too slow.

No, really. It's sort of black magic, Sha-256 uses 64 rounds. Nice even number. Different functions like Blake 2B uses 20 rounds. But then there's also a version that uses 12 rounds, which is faster. And people think, well, it's still seems quite secure. But if you want to be really secure, use the 20-round variant. If you want to be probably secure enough, use the 12-round variant. So there's no proofs. There's heuristics and things like that, and best practices. But this kind of cryptography is a little bit of black magic. And it's not based on any cool mathematical number theory stuff, either, the way that elliptic curve cryptography or RSA stuff is.

So if you break RSA, you can say, hey, I can now factor these composite numbers very quickly, that's, in and of itself, a cool mathematical discovery. The breaking of Sha-1, there's not really any cool math insight. It was just like, yeah, we found this fairly specific, weird path that we were able to break Sha-1 after a couple of years of computer. So it's cool, and some people are super into it. But it's something of a niche to actually build hash functions. I would recommend not building your own hash function. Yes.

AUDIENCE: I'm Wayne, and my question is, is breaking a hash function literally just guess and check, or is there more of a method to it?

TADGE DRYJA: So no. If you say, hey, I found a collision by doing two to the 128 attempts. One, nobody's done two to the 128 attempts. That's still seen as like beyond technology today. But if that's how you break the function, that's not really considered a break because that's sort of the definition, is yeah, well, we know this is 256 bits long. So to find a preimage, if you do two to the 256 attempts, you'll find it. So that's not considered a break.

A break is considered, hey, I found a preimage in two to the 240 attempts. Or I have a proof that you will be able to find a preimage in two to the 240 attempts, and here's how to do it. And that's considered a break. It's still impractical. Two to the 240's still impossible in today's technology. But if you had a paper and people looked at it, like, oh, yeah, that would work, you wouldn't be able to do it. But that's still considered broken.

And so something like MD5, MD5 output size was 16 bytes or 128 bits. So collisions, even if it were strong, it would still be too short today that collisions would be able to be found in two to the 64 iterations, which is doable on today's computers. If you run a bunch of stuff on AWS, you can do two to the 64 in a couple of days. But that's the different definitions of breaking the function.

Sort of fun. Ethan Hellman, who's at BU and we work with, he-- and we all broke the IOTA wrote their own hash function, which is like some cryptocurrency. And we found collisions in it. And it was kind of fun. But yeah, it was weird. It wasn't like number theory. It was just like, oh, well, I wrote this Python script and we have this go script, and we tried this thing and we got a collision. So it was kind of fun.

So usages. What do you use these hashes for? There's lots of cool things you can use them for. use them sort of as names or references, where instead of naming a file, you can just take the hash of a file. And that is a good, compact representation so you can point to what you're talking about . So the hash of a file is a unique representation. And if you change any bit in that file, the hash will change. And so you know that, OK, here's this way to point to a file. You can also use it as sort of a reference or pointer in different algorithms.

So you can say, anything you're using pointers for, linked lists or maps and stuff like that, you can say, well, I'm going to use a hash as a pointer and then be able to sort through it that way. So anytime you think of pointers and graph theory and stuff like that in computer science, think, well, could I use a hash function here instead of just like regular memory pointer? And in many cases, you can.

In some cases, you can't. So you can't have cycles. So the idea is you can't find preimages, you won't be able to find a-- whereas you could make a cycle of pointers in a computer, where A points to B, B points to C, C points back to A. You shouldn't be able to produce that with hash functions because having that cycle means, OK, well, somehow you found this preimage.

But in many cases, you can do this.

And another way to look at it is the hash is a commitment. You can say, well, I'm not going to tell you what x is, but I'll tell you what y is, and I can reveal x later. And then, since everyone remembers y, they can be sure that yeah, he's revealing the right thing. There are no collisions in this function, so we can be sure, if we're presented with x, that this was the x that was committed to yesterday. So I'll give a little example of that, of commit and reveal. So you can commit to some kind of secret or something you want to reveal later and reveal the preimage.

So here's my commitment. This is an actual hash, Sha-256. I just made it on my computer. And there is a string. There's an Ascii string that maps into this, and it is a prediction about the weather, but that's all I'll say. And given that information and given this hash, you probably can't find my prediction. You can try to try all these different Ascii strings about the weather today, but I'll reveal it.

So I think it won't snow Wednesday. But I think it actually-- anyway, and then I put this number in. And so if you put this in your computer in Linux-- I think in Mac it's a slightly different command. It's like Sha-2 or something. But in Linux, this will work, and you can say, I think it won't snow Wednesday.

And then I put some random numbers here because if I had committed to just the phrase, I think it won't snow Wednesday, you might have been able to guess that. You could say, well, he said it was about the weather. I'm going to take all sorts of millions of different strings related to days and weather and common English words, and I'm going to try hashing them and see if I find a collision. And you might be able to.

But I added this four bytes of randomness at the end to make that difficult. It doesn't really contribute to my commitment. And you know this doesn't really mean anything. But it makes it harder to guess what my input was because I've already revealed that it's not a fully random input. So you might be able to guess things. So I could say, hey, I'm going to make a prediction about the weather, commit to it, and then reveal my prediction tomorrow. And we'll see if I was right.

This can be useful in the case where-- not the weather, but in other things-- if knowing my prediction could influence the actual events, this would be a nice way to commit to what my prediction is without everyone knowing what the prediction is and then revealing it the next day. Yes.

AUDIENCE: What are the use cases for double hashing, like where you would hash that hash?

TADGE DRYJA: Hashing this again? Well, so in bitcoin they hash everything twice. Generally, you don't need to. There's no explanation for why they do that in bitcoin. You could. But there are things you can construct where you can, say, append some extra data and then hash this again. So you can say, here's my prediction for next week. And this is the hash, and then hash it again. So you can make chains of commitments and then reveal iterations of it.

Actually, I had some slides where you can sort of hash something again and again, and start revealing it incrementally. That might be useful. I actually have stuff like that in software. I've written where you want to reveal secrets. But let's say I want to reveal secrets, but I don't want everyone to have to store all of them. So I can make a chain of hashes, commit to the last one, and then as I reveal successive preimages, you don't have to store all of them. You can just store the latest preimage, and you can reconstruct all the hashes from that. Yes.

AUDIENCE: But is it computationally difficult to run double hashes?

TADGE DRYJA: So to evaluate-- if you want to try this, it's imperceptible. To perform one Sha-256 hash is, I don't know, a billionth of a second or something. You can generally do, like, 100 megabytes to a gigabyte of hash output on a regular CPU.

NEHA NARULA: I think she's asking does it make it harder to find a preimage if you hash twice, and the answer's no.

TADGE DRYJA: The answer's sort of no. It might. So I don't know, chained MD5, can you still find collisions? I'm not sure. But generally the thinking is, if the hash function is broken, and you can either find collisions or preimages, yeah, maybe it gets a little harder by iterating it. But you should just stop using it and use something that's secure.

But yeah, it seems that finding preimages would be harder since it's essentially adding more rounds by hashing it twice. And then there are some attacks, so it's fairly out there. But it's called length extension attacks due to how hash functions are constructed, where if you do say, OK, I'm going to take the hash and then take the hash of that, you do prevent certain types of attacks that are fairly niche. But a length extinction attack in a Merkle-Damgard construction will be prevented by this.

So generally, no. Generally, you don't need to do this. But there are different constructions where you're going to hash a bunch of times. I don't have the slides here but, like a Merkle tree is a binary tree of hashes where you're taking the hashes of these things and then hashing it again and again, and that's a really useful data structure. And a blockchain is essentially a chain of hashes. And that's what we'll talk about next week.

But yeah. OK. So I'm going to go a little faster. So that's an interesting use case where you can commit and reveal. And yeah, adding randomness so you can't guess the preimage. This is called a hash-based message authentication code where part of it is secret and part of it is not. And this is getting towards a signature, where I've committed to something, and then I reveal it, and everyone knows, yeah, that must be what he committed to the day before. It's not quite a signature, but it's getting to that direction.

And so next I'm going to talk about signatures. What is a signature? It's useful, and it's a message signed by someone. And so I'll define what a signature is through the functions that it uses. There's three functions will allow you to create a signature scheme, generate keys, sign, and verify.

And these different things. Generate keys, you make a secret key and a public key. And so the idea is there's some public key which is your identity , and there's some secret key which you only control. And you use that to prove your identity and prove that these messages are signed by you.

So yeah, you generate a key pair. The holder of the secret key can sign a message. And then anyone possessing a public key can verify a message signature pair. So I'll go into detail on these three functions. And this applies generally.

So I'm going to talk about a hash-based signature in detail, but there are many different signature schemes. DSA, ElGamal, RSA signatures, elliptic curve signatures. There's tons of different cool math systems that allow these kinds of functions. And I'll talk about in some ways, this is one of the simplest ones.

So yeah, there's these three functions. The first one is generate keys. And it returns a private key public key pair. And it generally doesn't take any arguments, but it takes in randomness. You need to flip coins. You need to find random one and zero bits. And it has to be long enough that no one else can guess what your private key is. So you have a private key, public key. Public key is public. You tell everyone. Private key is more secret key. Actually, I think in the code, I always say secret key. It's usually better to say secret key because at least it starts with a letter that's not p.

OK. And then the signing function, where you take your secret key and your message, and it signs a message and returns a signature. All these things are just strings of ones and zeros. It's just a bunch of bytes. Public key, a private key, a signature, a message. These are all just bytes.

And then the verify function, which is the most complex. A verify function takes a public key that you've seen, a message, and a signature. And it returns a Boolean whether this was valid or not. So it returns a single bit. If it's zero, it says, yeah, these two things don't match up. Maybe the message just changed, or maybe the signature has changed, or maybe it's from a different public key or something. But if all three of these are correct, and the signing function was the private key-- the secret key associated with this public key-- was signed to this message and produce this signature, then it will return true.

And so you get into the math properties of what does it mean to forge a signature, and can they be forgeable computationally? Eventually a lot of these things, since it's bits, you could eventually guess the forgery. But maybe that takes two to the 256 attempts or something.

OK. So any questions about the basic structure of what constitutes a signature scheme? Mostly make sense? And you can see how this is useful. You can publish a public key and say, hey, I'm Tadge. This is my public key. And in fact, on my business card, I have a RSA public key. And so if people get my business card and then I sign a message and email it to them, they could be sure that, oh, this is probably the same guy. Nobody ever cares.

But it's useful for the stuff we were talking about before with Chaumian cash, where Alice needs to authenticate to the bank, and one way to do it is to sign a message and say, hey, I'm Alice, give me a coin. And then Alice can sign a message to Bob and so on. So this is really useful as a basic building block for all these kinds of messages.

So I'll talk in the last 14 minutes about signatures from hashes. This is doable. Using just hash functions, you can construct a signatures key. And in fact, that's the first problem set. And you implement a signature system using only hashes. And the hash function is already defined for you. It's in the standard library. It's just Sha-256, the same thing bitcoin uses. And this is called Lamport signatures. Leslie Lamport wrote about this late '70s. I forget exactly when the paper came out. But this was one of the earliest cryptographic signature schemes.

And it's kind of cool. And another fun thing is it's quantum resistant. So if you know about quantum computers, quantum computers kind of ruin all the fun in terms of cryptography. All the cool things we can do with cryptography-- not all, but most of them get ruined by quantum. Computers but hash functions are quite resistant to quantum computers because they're not based on any fun math. They're based on this black magic of just XORing and shifting numbers around. That's a huge oversimplification.

But yeah, so those hash functions are generally seen to be quantum-resistance. So if you have a signature scheme that only uses hash functions, well, it still works, even if someone invents a quantum computer and can break all these other things, like RSA and elliptic curves. So there's actually renewed interest in these kinds of systems recently.

OK. So how do you make a signature scene with just hash functions? So how do you generate a key, in this case? So a public key and a private key you want to generate. So first we generate our private key. Now these squares are 32 bytes each, and you generate 256 of them on this row, 256 of them on that row. So you're generating 256 times two, or 512 32-byte blocks. And these blocks are each 256 bits or 32 bytes.

So in total, that's what, 8K? Eight kilobytes, I think. Pretty big. But anyway, you're saying, OK, here's my private key. It's all completely random. I just take slash dev slash urandom or whatever, just flip coins 8,000 times, or however many this is total, and generate all these different blocks and store them on my hard drive and keep it secret.

Then I want to generate the public key. So for each of these 32-byte blocks, I take the hash of it, which will also be 32 bytes. So there's now 512 hashes, 256 on this row, 256 on this row. The green will be my public key. And the gray one is my secret key. So they all look the same. They all look like just a bunch of random ones and zeros.

The gray ones actually are a bunch of random ones and zeros. The green ones are actually hashes, though, of all the gray ones. And I publish the green ones. Just to serialize it, I just put in a row. I say, OK, here's this first 32-bit, second, third, fourth, and then go to this row or whatever scheme you want.

So how is this useful? Now everyone knows a bunch of hashes, and I know a bunch of the preimages. So now it's sort of this commit reveal thing, where if I reveal to you this, you can verify that, oh, yeah, that mapped to this one later on.

Any questions so far about this process? Seems sort of useless but fairly straightforward. OK. Then I want to sign. So first, to sign a message, I'm going to take the hash of the message to sign. And this is often done. It's done in bitcoin. It's done in most signature schemes, where I want a fixed length number to sign. It's annoying to have to say, well, what if I want to sign a megabyte long file, or what if I want to sign of 10-byte long string? You want to standardize it. So whatever I'm signing, it's always 256 bits long.

So if I want to just sign the message hi, first I take the hash of the message hi, which in Sha-256, this is the hash of hi. And so I look at this as 256 bits, and I say, OK, I'm going to pick the private key blocks to reveal based on the bits here. So the first bit here is a one, because it's an 8. And so I'll reveal. And I indicated before that there's this zero row and this one row. And now what that means is, well, the first bit of my message to sign is a one. So I'm going to reveal this gray square.

And the next bit, the next four bits, actually, since it's an eight, are going to be zero. So I'll reveal this and then I'll reveal this, this, and this. And I just made it up. But yeah. So for example, if I'm signing, and it starts with 01101110, I reveal this preimage, this preimage, this preimage, this preimage, these three, this one. And so I reveal preimages based on the bit representation of the message I'm trying to sign, and then give everyone these.

So my signature will just be this sequence. I can go in row order here. Yeah, it's probably a lot easier. So I go in sequence. I say, OK, here's the first 32 bytes of my signature. Here's the next, here's the next, here's the next. And so my signature ends up being 256 blocks long, each of which are 256 bits. So it's like 8K. The keys are 16K and this is 8K or something. Fairly big but totally doable on a computer today. Eight kilobytes is no big deal.

OK. Now to verify, take the signature, hash each block of the signature, and see that it maps into that part of the public key. So the people who are verifying the signature, they have your public key. They have all the green squares. And now they have been given a signature, which is these gray squares, and they say, OK, well, let me hash this one. Oh, it maps to that, so it maps to a zero. Oh, this maps to a one, this maps to a one, this maps to a zero. And they can go through and say yeah, this is a signature on that message.

In the case of Lamport signatures, you can actually determine what the message is just from the signature in the public key. If you're given this and you're not told whether it's a one or a zero, well, just compare. Hash it and compare to these two green ones. You'll be able to see. And that's a useful signature because no one can forge that because no one knows these preimages except for the person who holds the secret key. So given your public key, I can't forge a signature from you.

Once the signature is issued, I also can't forge a signature. The only bit sequence I know is the one that you revealed. And so I know part of your private key. I know half of it. But that half only lets me sign the message you just signed. So I can't really do anything extra with this. So this is a usable signature scheme. I think I just showed it. But any downsides that you can think of with this?

AUDIENCE: You can only sign one.

TADGE DRYJA: Yeah, you can only sign one. Is that what you were--

AUDIENCE: You could also send the same message on to someone else with different signatures.

TADGE DRYJA: Yeah, but signatures are sort of public. So yes, you're saying that you can sign a message once and give it to a bunch of people. And that's sort of a feature, not a bug, I guess. There are different signature schemes where you want, I only want this signature to be valid to this person. There's different ways to do that with Diffie-Hellman key exchange and stuff. But the signature scheme we've talked about here with these three functions, the public key is really public, and anyone can verify. And that's something we want. If you don't want that, there's other ways to do it.

But yeah, the big one is, wait, you can only sign once. Once you generate a key pair, your private key, your public key, and you tell everyone these green squares, if you're try to sign again, you will reveal more pieces of your private key. So if I sign two different messages, sometimes it's the same bit. Sometimes it's different bits. And now I start revealing more pieces of my private key. And now people can start to forge signatures because I can say, OK, well, the first bit, I can sign anything on the first bit. I'm still constrained here and here and here. But in several locations, I can sign whichever bit I want.

And so the basic thing is, if there's one signature, I can't forge anything. If you give me two signatures, since it's generally random, on average, half of the bits of the signature will be constrained. So in this case, if it's 256 bits long and you sign twice, I probably still can't forge anything because 128 bits, I have the freedom to pick either. And the other 128 bits, I'm stuck with the one or the zero and I don't get to choose.

So that means most messages I want to sign, I won't be able to because if I tried two to the 128 attempts, I'll be able to find a forged signature. But that's a lot. And so maybe you can sign twice. But again, it's probabilistic. You might get unlucky and reveal quite a bit more than 128 bits, where you get both.

But on average-- and then once you have three signatures, OK, now I've probably revealed 3/4 of the locations you're going to have both the one and zero row. And you can start-- and this starts to be practical because in this case, you'd need a 2 two the 64 attempts to forge a signature. And that's doable on today's computers.