Bitcoin and Cryptocurrency Technologies

Week 1

Cryptographic Hash Functions

<https://www.coursera.org/learn/cryptocurrency/lecture/gFEJL/cryptographic-hash-functions>

In segment 1.1 we're going to talk about cryptographic hash functions.

We'll talk about what they are, and what their properties are.

And then later we'll move on and talk about what their applications are.

So, a cryptographic hash function is a mathematical function.

And it has three attributes that we need to start with.

First of all,

a hash function can take any string as input, absolutely any string of any size.

It produces a fixed-size output,

we'll use a 256 bits in this series of lectures, cuz that's what bitcoin does.

And it has to be efficiently computable, meaning given a string,

in a reasonable length of time, you can figure out what the output is.

So that's a hash function, but

we're going to need hash functions that are cryptographically secure.

The cryptographic properties of hash functions are a complicated

topic in general.

But we're gonna focus here on three particular properties.

And I'll explain in a minute what those are.

Start transcript at 53 seconds0:53

In particular, that the function is collision-free,

that it has a hiding property, and that it's puzzle-friendly.

And for each of these, I'll talk about what the property is, what it means.

And then I'll talk about why it's useful to have a function that has that property.

So, first, collision-free.

So, the first property that we need from a cryptographic hash function

is that it's collision free.

And what that means is that it's impossible, nobody can find values x and

y, such that x and y are different, and yet

the hash of x is equal to the hash of y.

Start transcript at 1 minute 25 seconds1:25

And so if we look at the operation of the function as depicted

by one of these red arrows.

Here's x and H(x), and here's y and H(y).

Then nobody can find a situation like this.

That you have an x and y that are separate, and yet when you hash them,

they hash to the same value.

Now one thing to notice is that I said, nobody can find.

I didn't say that there is no collision,

because if you think about it there has to be a collision.

Collisions do exist, and to understand why that is, we can use this diagram.

Over here on the left, I'm depicting all of the possible inputs to this function,

which can be a string of any size.

And over here, I have all of the possible outputs,

which has to be string of 256 bits in size.

So the right hand side here, the outputs, there are only 2 to the 256 possibilities.

Over here, there are more possibilities.

And so if you think that every point here on the left is gonna

be mapped by an arrow, to some point on the right.

You can see that as you go from all the points over here on the left into

the right, it has to get crowded.

And in fact, that there have to be multiple

values over here on the left that map to the same output over here.

In fact, in general, there will be a very large number of possible inputs

that map to any particular output.

So collisions do exist.

I said before nobody can find a collision.

And that's the key question.

We know collisions exist.

The question is are there any collisions that are findable by regular people

using regular computers?

Start transcript at 2 minutes 50 seconds2:50

Okay, now to make things even worse,

I said that it has to be impossible to find a collision.

Let me tell you how to find a collision,

because there's a method that's guaranteed to work.

And the method works like this.

That we're gonna pick 2 to the 130 randomly chosen inputs,

over on the left cloud of that previous diagram.

And if we pick those 2 to the 130 randomly chosen inputs,

it turns out there's a 99.8% chance that at least two of them are going to collide.

And so this is a simple method for finding a collision.

It works no matter what the hash function is, but of course, the problem is,

that this takes a very, very long time to do.

You have to compute the hash function 2 to the 130 times.

And that's, of course, an astronomical number.

This method works no matter which hash function we're using.

There's still a 99.8% probability that this works.

And if it doesn't work, just try it again, it'll probably work the next time.

But, this doesn't really matter.

And the reason it doesn't really matter, is that this procedure takes 2 to

the 130 steps, in order to get to that high probability.

So, we can say something like this.

We can say that if every computer ever made by humanity

was computing since the beginning of the entire Universe up to now,

the odds that they would have found a collision is still infinitesimally small.

So small that it's way less than the odds that the Earth will be destroyed by

a giant meteor in the next two seconds, which didn't happen.

Start transcript at 4 minutes 14 seconds4:14

Okay, so we know how to find a collision.

But this method takes too long to matter.

The question is, is there some other method that could be used on a particular

hash function, in order to find a collision?

And that's the question that is harder to answer.

Is there a faster way to find collisions?

Well, for some possible values of hash functions, of course there are.

For example, if our hash function were to simply take the input,

modulo 2 to the 256, that is, it just selected the last 256 bits of the input.

Then we would know an easy way to find a collision.

One collision would be the values 3, and 3 plus 2 to the 256.

So, for

some possible values of the hash function, it's very easy to find a collision.

For others, we don't know.

Start transcript at 5 minutes 1 second5:01

Now, one thing I need to note is that there's no hash function

in existence which has been proven to be collision free.

There are just some that people have tried really, really hard to find collisions and

haven't succeeded.

And so we choose to believe that those are collision free.

Okay, now, what good does collision freedom do us?

If we can assume that we have a hash function that is collision free,

then we can use that hash function as message digest.

And what I mean by that is the following.

That if we know that x and y have the same hash,

then it's safe to assume that x and y are the same.

Because if someone knew an x and y that were different, that had the same hash,

of course, that would be a collision.

Since there's not a collision that we know of, then knowing the hashes are the same,

we can assume that the values are the same.

And this let's us use the hash as a kind of message digest.

Suppose, for example, that we had a file, a really big file.

And we wanted to be able to recognize later whether another file was the same

as the file we saw the first time, right?

So one way to do that would be to save the whole big file.

And then when we saw another file later, just compare them.

But because we have hashes that we believe are collision free,

it's more efficient to just remember the hash of the original file.

Then if someone shows us a new file, and claims that it's the same,

we can compute the hash of that new file and compare the hashes.

If the hashes are the same,

then we conclude that the files must have been the same.

And that gives us a very efficient way to remember things we've seen before and

recognize them again.

And, of course, this is useful because the hash is small,

it's only 256 bits, while the original file might be really big.

So hash is useful as a message digest.

And we'll see, later on in this lecture, and in subsequent lectures,

why it's useful to use hash as a message digest.

Start transcript at 6 minutes 49 seconds6:49

So, the second property that we want from our hash function is that it's hiding.

And the property that we want is something like this.

That if we're given the output of the hash function,

that there's no feasible way to figure out what the input x was.

The problem is that this property doesn't exactly hold.

And to understand why that's the case, let's look at this example.

Start transcript at 7 minutes 10 seconds7:10

So here, what we're going to do is an experiment where we flip a coin.

And if the result of the coin flip was heads,

we're going to return the hash of the string "heads".

And if the result was tails, we're going to return the hash of the string "tails".

Start transcript at 7 minutes 24 seconds7:24

And now we're gonna ask someone who didn't see the coin flip, but

only saw this hash output, to figure out what the string was that was hashed.

That, of course, is going to be easy.

It's easy in this scenario to find what the input string was, it's easy to find x.

You simply compute the hash of the string "heads" and

the hash of the string "tails", and you see which one you got.

Start transcript at 7 minutes 45 seconds7:45

And so, in just a couple of steps, you can figure out what x was.

So the reason this example failed,

that is the reason why an adversary was able to guess what the string was,

was that there were only a couple of possible values of x.

Start transcript at 8 minutes 1 second8:01

And so, if we're gonna have a hiding property like this,

it needs to be the case that there's no value of x which is particularly likely.

That is, x has to be chosen from a set that's, in some sense, very spread out.

So that this method for the adversary of just trying all the possible values of x,

or just trying few values of x that are especially likely, is not going to work.

So the hiding property that we are going to need to set up

is a little bit more complicated.

And the way we're gonna fix this problem with the common value x, like heads and

tails, is we're gonna take the x.

And we're gonna put next to it, we're gonna concatenate with it, a value, r,

which is chosen from a distribution that's really spread out.

Start transcript at 8 minutes 43 seconds8:43

And so this H(r | x),

that means take all the bits of r, and put after them all the bits of x.

And so what we're going to say is given the hash of r together with x,

that it's infeasible to find x.

And that this will be true in the formally stated property that,

if r is a random value chosen from a distribution that has high min-entropy,

then, given H(r | x), it's infeasible to find x.

And what does high min-entropy mean?

Well, it captures this intuitive idea that r is chosen from a distribution that's

really spread out.

And what that means specifically is that there is no particular value that r could

have had, that would occur with more than a negligible probability.

So, for example, if r is chosen uniformly from among all of the strings that are 256

bits long, then any particular string was chosen with probability 1 in 2 to the 256,

which is truly a negligible value.

So, as long as r was chosen that way,

then the hash of r concatenated with x is going to hide x.

And that's the hiding property that the hash function will be deemed to have.

Okay, now let's look at an application of that hiding property.

And, in particular, what we wanna do is something called a commitment.

And this is kind of the digital analogy of taking a value, a number, and

sealing it in an envelope, and

putting that envelope out on the table, where everyone can see it.

Now, when you do that, you've committed to what's in the envelope.

Start transcript at 10 minutes 9 seconds10:09

But you haven't opened it, it's secret from everyone else.

Later, you can open the envelope and get out the value, but it's sealed.

So commit to a value and reveal it later.

We wanna do that in a digital sense.

So, to be more specific about what is the API that we're going to provide here,

the commitment API looks like this, that there are two things you can do.

First, you can commit to a message.

And that's going to return two values, a commitment and a key.

Think of the commitment as the envelope that you're gonna put on the table, and

the key as a secret key for unlocking the envelope.

Then later, you allow someone else to verify, given the commitment and

given a key, which you've told them in the meantime, and the message.

So they can verify that that commitment, key, and message really do go together.

And this will return a true or false.

Okay, now to seal an msg in an envelope, what we do is we commit to the message.

And that returns a commitment and a key, and then we publish the commitment.

That's putting the envelope on the table.

Now, later, to open the envelope, what we're going to do is publish the key and

the message that we committed to.

And then anybody can use this verify call,

with the commitment that we published previously, the key and

message that we just announced, to check the validity of our opening the envelope.

Okay, and the property, of course, we want from this,

is that it behaves like sealing an envelope.

And, in particular, the two security properties are these.

First, given com, the commitment, the envelope on the table,

that someone just looking at the envelope can't figure out what the message is.

Start transcript at 11 minutes 39 seconds11:39

The second property is that it's binding, that when you commit

to what's in the envelope, you can't change your mind later.

That is, it's infeasible to find two different messages,

such that you can commit to one message, and

then later claim that you committed to another, and the whole thing will verify.

Start transcript at 11 minutes 54 seconds11:54

Okay, so how do we know that these two properties hold?

Well, first we need to talk about how we're actually gonna implement

commitments.

And the way we're gonna implement commitments is like this.

Start transcript at 12 minutes 3 seconds12:03

That in order to commit to a value message,

we're going to generate a random 256 bit value and call it the key.

And then we're going to, as the commitment,

return the hash of the key concatenated together with the message.

And as the key value, we're going to return H of this key.

And then later, to verify, someone is going to compute this same hash of

the key they were given, concatenated with the message.

And they're gonna check whether that's equal to the commitment that

they saw, okay?

So this is a way of using hash functions of both in the commitment and

in the verification.

So now the security properties.

If we go down to the security properties that were at the bottom of the previous

slide, and we just plug in the definitions of how we're going to implement this here.

That is, this used to say com, given com infeasible to find msg,

we just plug in what com is.

Com is the hash of key concatenated with msg.

And similarly down here, this is what happens when we take

what was written there before and plug in the definition of verify in com.

Okay, so now what these properties become, the first one is given H(key | msg),

it's infeasible to find msg.

Well, it turns out that that's exactly the hiding property

that we talked about before.

Key was chosen random 256-bit value.

And therefore, the hiding property says that if we take the message, and

we put before it something that was chosen from a very spread out distribution,

like I said a random 256-bit value, then it's infeasible to find the message.

So this is exactly the hiding property.

And this one down here turns out to be exactly the collision-free property.

So that if someone can find two messages which have the same hash like this,

well then they have an input value here and

an input value there that are different, and yet those have the same hash.

And so because of the two security properties we've talked about for

hashes so far, this commitment scheme will work,

in the sense that it will have the necessary security properties.

Okay, so that's the second security property of hashes, that they're hiding.

And the application of that is commitments.

The third security property we're going to need is that they're puzzle-friendly.

And this is, again, a little bit more complicated,

but let me just go through it bit by bit.

That for any possible output value y that you might want from the hash function.

We're going to use y as an output value of the hash later.

That if k is chosen from a distribution that has high min-entropy.

That is, k is chosen randomly from some set that's super spread out.

Then there's no way to find an x, such that the hash of k and x is equal to y.

So, what this means is basically that if someone wants to target the hash function,

if they want it to come out to some particular output value y.

That if there's part of the input that is chosen in a suitably randomized way,

that it's very difficult to find another value that hits exactly that target.

So the application we're going to use of this,

is we're going to build a search puzzle.

And what that means is we're going to build a mathematical problem,

which requires searching a very large space in order to find the solution.

And where there's no shortcuts,

a way to find a good solution, other than searching that large space.

That's a search puzzle.

To be more specific, the idea is that if we're given a puzzle ID,

which is chosen from some high min-entropy distribution.

That is some very spread out probability distribution.

And we're given a target set, Y,

which someone wants to make the hash function fall into.

Then we wanna try to find a solution, x.

So that if we hash the puzzle ID together with the solution X,

we get a result that's in the set Y.

So the idea is Y is a target range or a set of hash results that we want.

ID specifies a particular puzzle, and x is a solution to the puzzle.

Start transcript at 15 minutes 57 seconds15:57

And the puzzle-friendly property here implies that there's no solving strategy

for this puzzle, which is much better than just trying random values of x.

And so if we wanna pose a puzzle that's difficult to solve, that we can

do it this way, as long as we can generate puzzle IDs in a suitably random way.

And we're going to use that later when we talk about bitcoin mining.

That's the sort of computational puzzle we're going to use.

Okay, so we've talked about three properties of hash functions and

one application of each of those.

Now let me talk just very briefly about the particular hash function we're going

to use.

There are lots of hash functions in existence, but

this is the one bitcoin uses, and it's a pretty good one to use.

It called SHA-256 or SHA-256, and it works like this.

Basically, it takes the message that you're hashing, and

it breaks it up into blocks that are 512 bits in size.

The message isn't gonna be, in general, necessarily exactly

a multiple of the block size, so we're going to add some padding at the end.

And the padding is gonna consist of, at the end of the padding,

a 64 bit length field, which is the length of the message in bits.

And then before that, it's gonna consist of a one bit,

followed by some number of zero bits.

And you choose the number of zero bits so

that this comes out exactly to the end of a block.

So once you've padded the message so

that its length is exactly a multiple of the 512 bit block size,

you then chop it up into blocks, and you then execute this computation.

You start with the 256 bit value called the IV.

That's just a number that you look up in a standards document.

And then take the IV and the first block of the message.

You take those 768 total bits, and you run them through this special function,

c, the compression function, and out comes 256 bits.

You now take that with the next 512 bits of the message,

run it through c again, and you keep going.

Each iteration of c crunches in another 512 bit block of the message and

mixes it in, sort of logically to the result.

And when you get to the very end,

you have consumed all of the blocks of the message plus the padding.

The result is the hash, that's a 256 bit value.

And it's easy to show that, if this function, c, this compression function

is collision free, then this entire hash function will also be collision free.

The other properties are a little bit more complicated, so

I won't talk about them here.

Start transcript at 18 minutes 15 seconds18:15

Okay, so we've talked hash functions.

We've talked about what hash functions do.

We've talked about three properties of hash functions and applications of

those properties, and the specific hash function that we use in bitcoin.

In the next lecture segment,

we'll talk about ways of using hash functions to build

more complicated data structures that are used in distributed systems like bitcoin.