[SQUEAKING]

[RUSTLING]

[CLICKING]

ROBERT TOWNSEND: This is lecture 9 on encryption. Be prepared. There's a bit of math in this, but the main goal is the conceptualization of the problem.

So we've got encryption is a tool in and of itself, and it can, therefore, be separated from the ledgers and so on. I have a slide on that. And then we'll come to three interrelated concepts, encoded message systems, hashes, and cryptographic puzzles.

And then I'll circle back to modern encryption, which uses public and private keys, giving you example of cyclic rings. And then we'll go into fully homomorphic encryption, multiparty computation, and zero-knowledge proofs in terms of what they are and some notation.

I have been threatening to go back periodically to Bitcoin, which keeps popping up in various of the lectures. In this case, we'll now hopefully have enough of the tools to understand the data storage system in so-called Merkle trees, or hashed trees associated with Bitcoin and the use of proof-of-work algorithm to randomize who is the validator.

And finally, I'll summarize how these FHE, Fully Homomorphic Encryption, multiparty computation, and zero-knowledge proofs fit together in terms of when they are needed or not.

So the point of this slide is something I mentioned last time, that programmed contracts are distinct from distributed ledgers. We can use encryption for both, and it's the third leg, in the sense, that we've covered already, distributed ledgers and smart contracts.

Encryption can be used as part of the validation protocol. Validation of the ledgers allows the financial counts to be altered with transfers of money and assets, but done in a secure way because it will be encrypted, hence, no trust is necessary, and this will create immutable and readily indexed records. So that's the ledger part of it.

As part of the programming of contracts, we can deal with all kinds of obstacles to trade-- private information, limited communication, limited commitment, confidential data. So the data need to be secure, and it can be costly to store and retrieve data potentially.

So encryption is a way to implement optimized solutions for two or more multilateral agents who want to use or solve a mechanism design problem, but they want to keep the messages they're sending back and forth secret. They want, again, the data to be secure and immutable.

They want to be able to commit to the arrangements and commit to the way they're implemented without any inconsistencies even if one-- or if you can wrap your mind around it, if there were two, even if both potentially want to renege.

So that's the big picture. Now let's go to the communication problem and basically think about encrypting messages. So there will be-- and I'm going to do this as kind of an overview and go through the other concepts, and then we'll circle back to some helpful notation.

So a public key is an object in the context of a exactly specified mathematical space, and the public key is used to encrypt messages so that what one sees are ciphertext. Gobbledygook.

The ciphertexts are public. There's no attempt to conceal them. The private key is used only for decryption of the messages. And the algorithms make it virtually impossible to decipher an encrypted message without the private key that was used as part of generating it.

There's an analogy here associated with prime numbers, and I'll keep coming back to this. And actually, it will be part of the mathematics in the end, that if you have two prime numbers, and the number-- those primes are large enough, it is very difficult to decompose into the underlying primes from the product.

And if you're thinking of 3 plus 5, that doesn't seem terribly hard, but as the numbers get larger and larger, it gets more and more challenging. So anyway, the analogy here is the private key is like one of the two factors that were used for the product, and it's pretty straightforward, computationally much easier to verify that you have a solution once one of the factors.

So there's a word in here, algorithms on designated spaces make it virtually impossible. That's a scary word because actually, you could get lucky and decipher things. It's just the odds are low. And also, as you'll see, it does-- it is a computational issue. So some of these better-known ways of encrypting are vulnerable to quantum computing, and we'll touch on that as we-- so that it's so not so hard after all.

OK, so that's messages. What are hash function? A cryptographic hash function is an algorithm that takes an arbitrary amount of an input in size and context and produces a fixed-size output called a hash. Some properties. A given input always generates the same hash value. The enciphered text is the same if the input is the same. And likewise, a hash cannot be deciphered except by random guessing.

So a good hash would be one that makes it very hard to reconstruct the original input from the output. This is called one-way function or non-reversibility. With just one change in one bit of the original input, the output-- the hashed output changes significantly and unpredictably.

So any tinkering is detected because you can compare one hash to a claimed identical copy, and if it used a different input, it will be different, and I'll come back to that. This anti-tinkering is called the diffusion or avalanche effect. And that's going from the hash backwards. Likewise from the inputs, if you had two different inputs to the hash, then it should be hard to find two different inputs that gave the same hash.

Now that reminds me to say that the power of this thing is that the hash is of extraordinarily low dimension relative to the inputs that are being used. And as you'll see, this is how Bitcoin, for example, is able to keep track of all these transactions since the genesis state.

So it almost seems impossible, but it's based on algorithms that are very hard to disentangle. And non-predictability means the input isn't going to tell you anything about the output.

So Sam generated these examples in the last few days to try to be concrete. Here's the text. This is an example document that's going to be hashed. The quick brown fox jumped over the lazy dog. You put it in this standardized hash function and you get this ciphertext.

But suppose you slipped and there's a little typo in this word over, and you get-- you can scan this for a while and try to see some similarity in the ciphertext, but there is none, that's the point.

AUDIENCE: But the reason it's not invertible is because we don't know what this function is exactly doing as people as us, outsiders? Or why is it not invertible, right? Well, it's actually not invertible in the sense that there's many things that map to the same output.

ROBERT TOWNSEND: I mean, you'll see some math momentarily. It's like basically private and public keys, and you can know a lot as the public keys, but the private key is randomly drawn, so unless you're lucky at guessing and you don't know the random key, then it's computationally almost impossible.

Here's an example of hashing an accounting statement as representative database, two people. They're always Bob and Alice, it seems, with their balances and their ID numbers, and that generates this-- and these are all of the same length. I mean, hashes can be longer than this, but this is illustrative.

OK. So hashes are like a signature system, too, for messages and for documents, data sets. If you have the original data set stored away on a server, then you hash it up and state exactly how it was hashed, and it's public. The hash version is public, but doesn't reveal the data.

Now, a new person comes along who does have permission to look at the data, wants to make sure that the candidate file they're about to use with real data is actually the data file-- the original data file and it hasn't been corrupted or altered. So if the original data are assigned by this unique hash, then the candidate file can be hashed up and compared pretty readily to the original, so you could spot discrepancies.

And relatedly, it's like sealing something, like stamping a document. You can-- again, by analogy with prime numbers, when they're multiplied together, it's really hard-- it's the hash, and say it's impossible to factor, then you have a document and you seal it with this product of primes.

So someone comes along and says they created the document, that's my signature, and they give a validator an external third party one of the two factors. And now it's relatively easy for the external party to verify-- to totally factor the seal, as it were, and verify that it's the same.

So by stamping it, the author of the document or the owner of the data is making a commitment to authorship. So in this case, you don't actually have to decompose it. If it shows up with a seal, then you could take it as given that it's going to work. The verifier could potentially, but not necessarily always verify the seal is authentic, that the document is authentic.

So it's a commitment. You can think of it as solving a commitment problem. If you have a timeline of the creation of things along the way, and then use later by third parties and so on, you can't go back and say, oh, oops, that wasn't really my document.

The third part essentially we've already mentioned, but it's good to think about it separately. Cryptographic puzzles. The problem could be hard, but not too hard, to solve the puzzle, and you can control the degree of hardness, and it is then subsequently solved by trial and error, but how many trials on average and so on and how much time it would take is, again, subject to control.

So prime numbers are like that. They could be large, but not too large. It takes time and energy to-- and computational power to factor the product. So you can parameterize the difficulty. And that is one of the better-known parts of Bitcoin's validation protocol, and I'll come back to that about three-quarters of the way through this lecture.

But a little bit more on factoring primes. First of all, in a subsequent slide, there's something about two numbers being called relatively prime. That means they're kind of prime for each other. They only share-- they don't share a common denominator other than unity.

The main point here is wrap your mind about large prime numbers. 3,457,631 and, looking at digits, 4,563,000, et cetera, are each prime numbers. And if you multiply them together, you get this guy, which, if I did it correctly earlier this morning, is a billion-something.

I mean, it's kind of amusing. Like, there is something known as the largest known prime number. And presumably it's not the last one, but that was the latest-- and it's all related to computational demands to, say, find a new prime, or in this case, decompose these products and so on.

Now decompose has to do with-- is not this-- if you claim you have a solution, then you can verify that it is a solution. Finding a prime factorization of a given number with n digits is believed to be of exponential complexity, although that's not known for sure.

The latest known algorithm is 10 to the power square root of n log n, essentially exponential, but the verification procedure, when you have a solution to verify that it is a solution, that's polynomial. So it's easier to verify stuff once you have a guess than it is to do the actual factoring to begin with.

So again, the problem of factoring large integers is believed to be of exponential complexity. The current RSA encryption method is premised on that hypothesis, so it's not a known fact, you might say. It's just there's no evidence to the contrary yet.

If we could find a way to factor large integers in polynomial time, then RSA-encrypted messages would be readily decrypted, deciphered. So this is the spirit of getting into the vulnerability of these algorithms in terms of the premises that underlie them, but also ultimately quantum computing.

So back to the big picture. Encryption, in this case, thinking about messages, we may take it for granted now, but it's relatively recent that we have this breakthrough due to Whit Diffie's ideas.

Previously, the same key machine that scrambled the message would also be the way to decipher it. I mean, for me, I always think of World War II and the enemies were encrypting messages, and there was a big value in getting the cipher machine off the enemy boat because then you could use it to-- and that's the deciphering machine, say, then you could decipher all the messages.

But the breakthrough here is-- I guess that's what it says here. The very tools that eavesdroppers lusted after, the decryption keys were passed from person to person, they existed in two places, so it was very, very hard to basically keep them non-public.

But under Whiffie's design, this deciphering is done by the public key and it is public. Everybody can know it, it's published. Widely disseminated.

So instead of keeping everything secret, almost everything is public. All the encrypted ciphertexts are presumed to be public. The functional forms and spaces used for encryption are public, but not, of course, the messages themselves, which need to be kept secure, and in particular, not the private key, which has to do with the decryption. So public key for encryption, private key for deciphering.

So finally, let's get to a little-- I myself have trouble understanding things if I don't see a bit of the math, so let me share with you some of that.

So if we have a group of objects, say group G, we can take two any elements and add them or multiply them. So addition and multiplication are binary operations on two elements of a space. In well-defined spaces, any two elements can be added together or multiplied together.

As long as certain key properties are satisfied, then this space G is referred to as a ring. The properties are going to be familiar for the binary operation of addition. It has to be associative, so you can put parentheses around stuff in different order. Commutative, you can just reverse A, D, D, A, have an identity element-- this is for addition, so it's like subtraction, the inverse to get the 0 element.

And for multiplication, the operator must be left- and right-distributed. Again, associative in some cases, whatever that means, is commutative and have an identity element, which is 1. So again, you think of inverses, 5 times 1/5 gives you 1. So an inverse is something that will map back to this if you find it to this identity element.

I mumbled something just now about "in some cases." It's really important to remember, there isn't just one space that people use there. An encryption technique consists of specifying, in this case, rings with their elements and so on, and their use. So maybe you just need additive encryption and not multiplicative encryption. But I'll show you one.

And this is a finite number of integers 1, 2, dot-dot-dot, n. Zn, or Z sub n, is the notation for a set of a finite integers of order n with the provision that two integers, when added, would give you a number. If that sum is larger than n, you divide by the sum. You divide the sum by n and you get the remainder. That's a modulus n or mod n

And I have-- but I haven't read very thoroughly, a document which is, what is this obsession with remainders? It seems like a lot of encryption is all about the properties of remaining elements after you divide. And you can think of them as a nuisance, like when you're dividing something by longhand, it's like, oh, now I have to add a whole other thing and do it all over again. So it's computationally-- that's where the computational aspects start entering in.

Multiplication is the same. Take any two elements of G termed the order, the order of a group G is the number of elements in G, and it's a finite number, as above. And a generator of group G is an element of G which, when multiplied successively under the multiplication operator, generates the entire space.

So we're going to do the multiplication operator r times-- if we do the multiplication operation r times on an arbitrary element of h, of G, then we can use this notation h to the power r, which means h, this particular element of G, is multiplied by itself successively r times.

Now, G is not necessarily this h. h is any element in the group. G is a particular element that's the generator, and there could be more than one. So Sam created this example. So this is a cyclic ring. The order of the group is 11. There are 11 elements in this group, including 0. And the generator is 6, and the modulus is 11. So basically, you take 6 and multiply it-- don't multiply it. r equals 1, so 6 to the 1 is 6, so you get 6 back.

If r is equal to 2-- sorry. Yeah. Then 6 times 6 is 36. 36 divided by 11 leaves a remainder of 3. So for r equal 2, n is equal to 3. So the outer circumference of this are the remainders after you take the modulus of the powered object. And the interior is the number of times you're multiplying by itself successfully.

Now, looking at this representation-- oh, and why is it a ring? Because for various values of r, you will get all of the 10 objects. That mean 0 is kind of a trivial object here, so don't worry too much about it. You get all the underlying elements in G.

And then he's staring at this, it's like, well, it's one to one, so how obscure is it? This is back to the cipher machine. If you know the code that generated this circle, and under Duffie's design, you would know the ring and the generator key and everything else, and if you saw this so-called randomized output, you could completely decipher it.

Now one thought from that was, yes, but we're going to be encrypting and trying to decipher things for h, not G, but nevertheless, I think some of the intuition carried over because then we find this, ElGamal, which is what we're looking at, cryptography, keeping the private key secret when the order of the cyclic is small-- so not so many elements-- is impossible.

An attacker could easily brute-force calculate the private key by trying out all possible values within the group order, and that's called baby step, giant step. I don't know why. It sounds like someone's on the moon. But anyway. So it's crucial that the cyclic group be a very large order, have a lot of elements.

So again, this thing about integers and primes keeps coming back. Things have to be big here for this stuff to be hard to solve. So as a review, partly, when this repetition operator r is random and concealed, then with the modulus, outcomes from repeated operations, put one back into a random and unpredictable element of the ring, and rings with generator G are called cyclic rings.

So for notation, r is the private key and h of r is the public key. So h is known, but r is kept private. Well, sorry, h of r is known, but r is kept private. So the space G with these operations and its modulus, its order, and the generator are all understood are public in addition to the public key. So that's an example in notation.

So let's see how it works with incoming messages. OK. So there's three components to doing this-- the key generation, the encryption, and the deciphering. So there's Alice and Bob. And Alice is the one who's going to receive the message from Bob. Alice generates a key pair. She chooses a cyclic group G with q elements and a generator g, and this has good properties, like it has an additive element.

She chooses an integer x randomly from the set of numbers up to the order of the group minus 1, and computes h equal g to the x. So h here becomes the public key. And in fact, all of this is published, not just h, but also the group, the order, and the generator. And Alice secures her private key. And for that to work, she's got to keep it secret. Don't hand it to a third party, put it on a server, et cetera, et cetera.

OK, so encryption. Bob wants to send a private message to Alice using Alice's public key. So the message m that Bob wants to send is mapped into the underlying space G, and that's not supposed to be hard to do. That's like ASCII and binary and Greek and everything else. You can go back and forth to the representation of objects. In this case, mapping it into G itself.

Then Bob now chooses another random number, y. Again from no larger than q minus 1. And Bob computes using h, which is known, h to the y, something called a secret-- or a shared secret. You could initially think that if Bob has chosen this at random, then it's not even going to be shared with Alice, but again, the point of this is that part of Alice's process to decipher the message m will reveal the secret to Alice.

OK, so Bob chooses this random object, randomly chosen object y. h to the y is the secret s. Bob sends two ciphertexts to Alice. g, the generator, to this random value y, and also sends the message multiplied with the secret. So Bob gets a vector. Two ciphertexts-- oh, sorry, Alice gets two ciphertexts.

So Alice takes those ciphertexts and she has her private key. Note, Bob never saw it. The first thing is, Alice can compute the secret. How is that possible? She has the ciphertext from Bob, which is g to the y. So c1 to the x is g to the power y times x-- and I'll come back to this in a minute, but h here was g to the x. So this g to the xy is h to the y.

Now a little bit of a comment here. In this representation of the encrypted space, it looks like you can do the usual operations on the powers of some-- but it's not necessarily true that in our normal spaces, g to the x raised to another power where the whole object is being raised is just the product of the two powers. That's not true, but it is true in these encrypted spaces.

So lest we lose the thread of this, Bob has sent the ciphertext to Alice. Alice takes the c1 ciphertext and actually computes s. So at this point, Alice actually knows y because she knows h and is seeing that part of the ciphertext.

But next, Alice computes s inverse because she knows s, and every element in G has an inverse. So Alice can decipher the message just by taking the second ciphertext and post multiplying by the inverse. So c2, the second ciphertext is m to the s. So c2 to the s inverse is just m times ss inverse, and you get m.

All right, it's a bit tedious on the one hand, but at least you can get a good sense of how the encryption system is working-- in this case, allowing Bob, using the published public part of the key from Alice, to send Alice a secret message that she can decipher, but no one else would know because, again, Bob is randomly choosing-- it doesn't say that line by line, but Bob randomly chose y, so Bob has his version of a secret private key.

OK. Signatures. This has to do with getting ready to do outgoing messages. Interestingly, incoming messages to Alice are easier, so be forewarned, than dealing with these outgoing messages.

So Alice wants to send something and she wants to sign it. That would allow verifiability of her identity-- she is Alice, and the message content came from Alice, so she's committed to be the author, and having sent the message. So again, no reneging if this works. Her identity is known, and the fact that she sent this message will be known. Not necessarily-- but anyway, this is the way a signature works.

The message is kind of separate. We're going to focus on signing the signature part here. So private keys are used to generate a digital signature for the outgoing message such that it can be verified who knows the signer's corresponding public key. So that's similar to what we just did. The public key is public and everybody knows it.

And the receiver can verify the origin of the message, authentication from Alice. The sender cannot falsely claim they have not signed the message, non-repudiation. And the message has not been modified since it was signed, integrity. So those are all really nice, strong properties.

AUDIENCE: For the previous encryption we just saw, we didn't-- we only needed a group and one operation and to exchange the powers? So that comes from associative or something from the group? I don't know.

ROBERT TOWNSEND: That's what was in the notation, yes.

AUDIENCE: So-- but then those are the only properties we really needed, right? We didn't need--

ROBERT TOWNSEND: I think it's probably true, yes. I haven't thought about that, only a subset of the properties. I mean, there was kind of a hint of that at the beginning. Let e represent the additive element.

AUDIENCE: Oh Yeah, it only needs one, OK.

ROBERT TOWNSEND: Yeah.

AUDIENCE: And then exchanging the powers, and then I guess we also need it to be commutative?

ROBERT TOWNSEND: Yeah.

AUDIENCE: Where do you do that?

ROBERT TOWNSEND: I don't know, it's not quite commutative. I mean, this thing here--

AUDIENCE: I mean, yeah. Maybe it's just associativity. It doesn't matter what order you do the operations.

ROBERT TOWNSEND: Yeah.

AUDIENCE: Or maybe it also needs to be-- I don't know.

ROBERT TOWNSEND: All right. But, I mean, I'm agreeing that--

AUDIENCE: Not everything is--

ROBERT TOWNSEND: Yeah. OK. So this is the Schnorr signature scheme for outgoing messages. And as I said, it's a little bit more complicated, but we'll give it a go. In this case, we're going to have three phases to it. Parameter generation, then a more familiar phase about generating keys, and then-- actually four. Assigning part and a verification part.

So the parameter generation, you choose a length of N and a prime-- N-bit prime number p. See, these primes are actually here. We choose a cryptographic hash function, H. It has more-- it's longer than what we need. We're only going to need the first N bits of it-- and this ended up with Bitcoin. Perhaps for this reason, I'm not sure.

We're going to choose a generator g less than the number p, the prime number p of the cyclic group of integers modulo q. This is a little bit of a complication and I'm not going to even attempt to explain why it's needed, but before, we were just in one space with a certain order and a certain modulus, and here, we have two objects going for us. We have p and q, and they're different from each other.

And all of this can be shared among the users of the system for key generation. We do the following. We choose a random-- randomly choose an integer from this set 1, dot-dot-dot, up to p minus 2, and then not surprisingly, we choose g, the generator, to this power A where A is randomly chosen. So A is looking like a private key. It's modulus p, not q.

And so as a summary, although I hate this notation, the secret key SK is little a. And the public key of this scheme is g to the a modulo p. Why do I dislike it? Because this looks like the private key, but it's the public key. The private part is the secret key. Anyway, just a nuisance. OK.

So the signing. There's a message m, and as I said, that's a bit separate here, but on the outer envelope, you might say, Alice does signing in the following way, and let's just jump down here. Alice picks a random number k, and she does the usual thing, which is the generator to the power of k mod p is here denoted as h as we have been doing before.

So it's effectively as if Alice is sending some randomized object to Bob h, which looks like a public key for purposes of signing. Now Bob gets this thing and Bob picks something else, a c, a constant, from the set 1 through q of integers, and Bob sends that back to Alice.

So then Alice takes c, that she just got from Bob, pre-multiplies by a, which is her private key. Adds k to it and takes the modulus q. And sends that to Bob. So if all these steps are followed in this way, Bob can verify the signature as follows.

I mean, Bob got s from Alice, so he can take g, the publicly known generator, to this power s, and he knows what that is because he has knowledge of every bit of it-- every piece of it. The question is whether the public key that was part of the scheme, when raised to c, multiplied by h, which he-- which Bob got from Alice in the first step, is equal to this object on the left-hand side.

Back to the thinking of the big picture. Bob knows all these objects as a outcome of this messaging that's going back and forth. Are these things equal? Well, if all these steps were followed, then g to the s is just g to the what s is, ac plus k.

And over here, this public key raised to the power c is like g to the a raised to the power c, which is like a product, and times h, but again, h was given by the first step that Alice did somewhere here. Alice signed it as h equals g to the k. So Bob knows that, too. And so you can now almost trivially see that these two objects are equal to each other.

OK, that's the most tedious thing we're going to do today. I'm a stubborn guy on some dimensions. I insist on going through the math so that we understand that this, more concretely, exactly how this algorithm is working in terms of signatures rather than just claiming that it works.

So this signature scheme will always be accepted by the receiver of the message. In this case, again, Alice is sending it to Bob, and Bob knows it must be Alice because only Alice could have encrypted things in such a way as to send it back to him that would have made this equation hold.

OK. Well so far, we've been talking about sets of integers modulo stuff, granted, of large dimension. But it turns out that this is driven by the concern that these schemes could eventually be broken by quantum computers, the encryption community has turned to polynomials.

So this part is the same. It was Z a minute ago. Z to the n, this is like F to the q, are these sets of integers that always include 0. In this case, when q is 1, F2 is basically 0, 1. F11, we kind of saw that a moment ago, 0 plus the first 10 integers, F23, et cetera. So we choose a field.

And then these numbers in a given field become the coefficients for a polynomial. So this, for example, is a polynomial where the largest power is n minus 1, and it has a constant term out here. So it has n terms, but n minus 1 powers.

And then these numbers, the little a's, are basically going to be determined by objects drawn from these fields. And where's the modulus? It's not so obvious from this slide, but you could have a polynomial like this one and multiply it, and we're going to get a much larger power, but the modulus is going to be another polynomial, like x to the n plus 1. So this is going to reduce the power, keep control over the highest power in the polynomial by virtue of division.

What's not on the slide here, although I was reviewing it this morning, is something called ring learning with error, and it builds on these polynomial schemes. So you have small polynomials, and you also have random polynomials. And you start generating the sums of polynomials with these-- have in addition the addition of these randomly chosen terms.

And it gets out of control really quickly. And in particular, computationally it kind of gets out of control depending on how many operations you're doing because you're throwing an error term, error polynomial into every operation. And so you could start with the error term being small, just enough to cloud up the true answer, but it can compound itself and get bigger and bigger.

So a lot of the modern encryption, as I understand it, is ways to put bounds on the size of the error term. Sometimes they use the word "bootstrapping" to mean they kind of stop the process and start taking out some of the accumulated error and then they start it up again and so on.

So hopefully you're happy that I'm not going through all of that. But the outcome of doing this is to generate something which is quantum-proof, quantum computing-proof. And an intuition for that has to do with having multiple generations.

Depending on the power term of the polynomial, you could think, for example, of having multiple agents in control of each one of the coefficients. So even if you crack the code on one of them, you haven't got all the other ones. So it's a kind of a decentralization that, for reasons I don't really understand, survives quantum computing. OK.

Next, some more general concepts, homomorphic encryption. OK, so the basic idea with homomorphic encryption is, again, the mathematics, you have an isomorphism between two structures, a mapping from one to the other; and the reverse, the other to one. You have direct and inverse mappings.

If the two structures are algebraic, these isomorphic mappings are termed homomorphic if it's one to one, so that every element in the domain has a representive in the range and vice versa.

So isomorphic mappings can preserve relationships, and that's going to allow us go from the true underlying message spaces to encrypted spaces being able to do operations on the encrypted space even you don't know what the objects are and preserve the outcome.

In particular, then, in terms of notation, we have, say, a message little m, and the mechanism design problem wants to operate it with a function f, like a risk-sharing contract or whatever. But the sender of the message may be worried that she'll reveal something, so it gets encrypted.

Encrypted value of m now operates on the encrypted value. It turns out that with this isomorphism, it's equivalent with the encrypted value of f of m. Both sides of this equation are in the encrypted space. So here, you wait till you have f and you encrypt it, or our real goal is to encrypt m, have f operate on it. Because it's already encrypted, f of the encrypted value will be in the encrypted space.

Thinking-- so this is true under homomorphic-- fully homomorphic encryption, another way to look at it is to decipher the left-hand side. So if you take m, encrypt it so it's private, have f operate on it in the smart contract, you get an encrypted number. You then decipher that number, and it's like an inverse operator, you're going to get-- let's look at it this way.

If you decipher the encrypted value of f of m, you're going to get f of m as the answer at the end. So you'll see f of m, but you won't see m. ElGamal, we've been going through, is an example of homomorphic encryption.

What is multi-party computation? Also called secret sharing or secure sharing. It refers specifically to having multiple participants. In this case, you may recognize these names, Andrew, Susan, Vinod. Andrew is Andrew Low. Vinod, last name is escaping me. So it's not deep science. S's denote the private data, the secret, so to speak. And the x's are these randomly chosen numbers that cloud up what the s really is.

I don't know whether it's a good or bad example. The secret sharing, people refer to people in a room, they're all really filthy rich, and they really want to know who the richest one is, but they don't want to reveal their wealth.

So anyway. So Andrew clouds up his wealth-- or his line item on a balance sheet by x. He sends it to Susan in this order. Susan takes that output, which is obscure to her due to the x, and adds in her secret information, and a new noise, and so on up the chain it gets to the last person who builds on the previous sum to add his or her-- his, in this case, secret plus noise.

And then-- so you have the sum of secrets and the sum of noises at this point. Then it goes back to Andrew. So Andrew says, well, I don't know what this object is, but I'll take out of it the noise that I put into it. And then Susan takes out of it the noise she put into it. And you end up with Vinod at the end, who can publish the result, which-- because all the noises are now eliminated, is the sum of the secrets.

It is the average wealth, so to speak, of people in the room, it's just this last number that Vinod publishes divided by n, n being the number of people here to get the average.

AUDIENCE: How do you know who--

ROBERT TOWNSEND: Yeah, I should have said the average. Well, at least you'd know if you're above or below the average.

AUDIENCE: That's true.

ROBERT TOWNSEND: OK. Now, this isn't costless. There's a lot of back-and-forth messages going on here among these agents. And in practice, it looks simple because we're not doing it in encrypted spaces, but-- OK.

So this we did, MPC we just did. Just want to say, you can operate on F2, you'd have a function g. You have m producing f, and then some g composed on f, and that's what you really want, and you encrypt it.

So this would be the outcome of an encrypted space, and that could be generated by, for example, having m encrypted into f and then operating on g, or, if you want, you could just have f of m running on the side in some private survey and then encrypt that and have g operating.

All three of these things are the same object, so when you start deciphering them, because these objects are in encrypted space, you get, for example, g of f m back without having ever seen m.

And I will say that next time, I'll go through examples-- economic applications, auctions and risk-sharing with private values and so on to see how multi-party computation and homomorphic encryption end up getting used.

OK. Finally, zero-knowledge proofs is another concept. How can a prover convince a verifier that she or he knows something without ever revealing what that knowledge actually is? And that could be needed when we have pre-existing accounts on ledgers and wish to keep some secrets, like transfers among the ledgers. I'll come back to that in the last slide.

So the idea here was zero-knowledge proof is for the one, party prover, to convince the other one, the verifier, that the prover knows something without ever revealing exactly what that knowledge is.

The worry is not about the prover, the worry is the verifier, if she or he learns something in the process, could do something evil with that information. There's an example here, but it's a bit misleading because, in effect, it's like giving up your secret key, which is more meant to be here to say this is not as straightforward to come up with these zero-knowledge proofs.

There are two examples. One is there's a pen which the prover knows is a special pen, and the prover can actually see something on the pen that's obscure to the verifier. And there are two pens, one with the special markings that the verifier can't see and the other one is bogus. And so the prover says, I know which is the real pen.

And so the pens are given to the verifier, and he or she changes the order or not in her hand, and puts them out and shows them to the prover, and the prover picks-- I mean, I'm sorry, I should have said the start of this thing is that the prover says that's the special pen, and the verifier says, OK, I have to take your word for it, then shows the pens, and it is the one that was previously designated.

Now that could have happened by chance, 50/50, that the left hand actually was the hand holding the special pen. So they do it again and again each time. The verifier does not know anything more, it's an independent experiment. But you can see, asymptotically, she would get convinced that the prover really knows which is the right pen, but she never ends up knowing.

Another example has to do with coloring nodes in a network in such a way that any edge connecting two nodes is supposed to be of a different color and you only have three colors. So this turns out-- I don't know if you covered this in your CS classes. This is a well-known hard problem, but someone claims to have the solution.

So again, it's like, OK, here's-- I have my map, you get to pick two nodes. So then that is revealed, and the edge is there, but they're of different color. OK, so maybe that was a fluky part of the network that was colored correctly, and so you do it again independently a different edge. The colors are changed in the process every time because once you have the solution, you can color pink versus yellow, doesn't matter. And so the verifier gets convinced.

Another-- it's called zk, zero-knowledge, proof is associated with Neha, who's over at DCI, and this applies for ledgers, are one of our favorite objects. So these are transactions on ledgers among Goldman Sachs, JP Morgan, and Barclays, transfers from one account to another, except that none of these three guys wants to reveal all of this, not even to the regulators.

Say the regulators only want to make sure that transactions are balanced, that the sum of what came in minus the sum of what went out from one of these accounts is zero. They're not double-spending. So how do you-- v is the value of each one of these transactions. Each one of those transactions is going to get obscured with two generators, g and h. One, g, takes the value v as the power. The other one, h, takes r as a arbitrarily chosen random number from ElGamal-type considerations.

And this product to powers is called a commitment. It's a Pedersen commitment. And in more detail, you have a cyclic ring-- sound familiar? With a certain number of elements. g and h are both generators in that group G. And a commitment to these values that are transformed to lie in the space G is done, again, by taking these products. A very useful property is that it's additively homomorphic. So in this case, it's specificly limited. They can get what they want with additivity, so what's going on here?

So you take these commitments, you take the product of two transactions-- that's another ciphertext. It's a commitment to the sum of the values, not the values individually. It's randomized by the sum of the random numbers. Basically commitment 1 is g to the v1, h to the r1; commitment 2 is g to the v2 h to the r2. You multiply those commitments together, which is equivalent with adding the powers. So you get the sum of the v's and the sum of the r's.

Now you could reveal the sum of the r's to the regulator, who would then be able to back out the sum of the values. Revealing the sum doesn't reveal the individual numbers. So the underlying individual transaction values are still concealed, but the regulator can be assured that there is no double-spending.

OK, so I promised to go back to Bitcoin. And again, we've got the picture of the Merkle tree of these values. So it's a block that are chained together. The blocks have a header on the top of, say, the structure. The rest of it has to do with the transactions, the number, the count. The header actually contains a lot of the information. It has the previous block ID, a timestamp, a nonce-- I'll come back to that, and a difficulty.

Again, we're using the hashing part. I alluded to this earlier. So you have all these individual transactions. Each hash, you take the hash of the sum of the two of them, another hash, and you make your way back to the latest route of all of it, all in a manageable database.

The leaves here are the transactions. Each node consists of hash of the children as you go back down the tree. And changing any transaction in any one of these blocks would change the route by virtue of the hash.

So where is the nonce? This hashing-- actually, double-hashing, you take the block header and you hash it up twice, and you end up with something-- a 250-bit block ID. And the nonce is one of the objects in the block header, so it will change. If it changes, it changes the double hash.

This nonce controls who wins the deciphering contest. Say the criterion for winning is that the number, after all-- the hashing should be smaller than a certain number. So for example, 2 to the 187. Now what is the likelihood of a 256-bit number being that small? The answer is, it doesn't happen very often.

And the math of it is-- from Sam is right here. What does it mean for something to be a 250-bit length? It just means 2 to the power i for each term i, 1. 2 to the 1-- 2 to the 0, 2 to the 1, 2 to the 2, et cetera, et cetera, and then add it all up at the end, that is equivalent with 2 to the 256 if every bit was 1. If every bit there turned out to be a 0, then you end up back in 0. So how do we get the first 187 bits to be 0? That's not going to happen very often. You can ignore this last line. Deb and I were exchanging slides.

So hopefully that helps a little bit in terms of controlling the degree of difficulty in Bitcoin in validating the transactions. They're all randomly putting in these numbers in such a way as to try to get the resulting output to be relatively small, and that doesn't happen very often, so it's all trial and error.

To summarize today a bit, this is something Nicholas found, it's really quite helpful. Can sensitive data be revealed after being used in a process? If the answer is sensitive data could be revealed, then you can use commit-reveal-type encryption such as the Pedersen commitment. Willing to reveal the sum of things, but not the individual values.

On the other hand, if the answer is can some sensitive data be revealed is no, then the new question is, can the use of data entirely consist of a series of proof statements, like is it true that we have this value f? Is it true or false when x is the confidential data and f of s is the statement? Is it true that f of x, this value, is that the true value or not? It's a true/false statement, it's binary.

So that could-- if that is the answer to the second question, is yes, then you can use zero-knowledge proofs. On the other hand, if the answer is no, then the new question is, can the use of data be managed by a single entity? And if the answer to that is yes, then you can use homomorphic encryption.

But if the answer to-- no, it can't be a single entity, then you have to split the data up into several-- among several entities and you need multi-party computation for that.

So here, you can see, depending on what you're trying to do and what the conditions of the application are, whether you can get by with Pedersen commitments, or zero-knowledge proofs, or homomorphic encryption, or-- and whether you need multi-party computation.

OK. OK. So I'm over by about three minutes. Thank you for your patience. Again, I do understand this is a lot of material, but that is an overview of modern-day encryption. I hope you found it helpful. And next time, we'll actually use it in some economic applications. Thank you.