SRINIVAS DEVADAS: All right. Good morning, everyone. So more of the same in terms of cryptography and cryptographic techniques similar to Tuesday's lecture. So if you liked it, you'll like this one. If you didn't like it, well, it's going to be more of the same, so sorry.

But what we're going to do today is do a couple of things a little bit differently. We're going to talk about encryption. So we talked about hashing, which, of course, you know about from the use of dictionaries. We haven't really talked about encryption in 046 or even in 006 previously.

But we look at two different kinds of encryption algorithms. We spend a little bit of time on symmetric key encryption, which is really something that was used in encoding machines including the enigma from World War II that you probably saw if you saw *The Imitation Game.* So there you have a shared secret. And it's that single, what's called, symmetric shared secret. Both parties know the secret.

And very quickly, we'll talk about what it means to actually exchange a secret. So we'll talk about key exchange. And then I'll move to asymmetric key encryption, which I alluded to a little bit when we talked about digital signatures last time. I talked about public keys and private keys.

But we're going to actually look at a couple of different public key encryption algorithms today. The classical algorithm, the very first algorithm really that stayed secure, the RSA algorithm. It stands for Rivest, Shamir, and Adleman, the initials of the three inventors, invented at MIT in 1977 and still in use today.

And the last part of today's lecture is going to be looking at hardness. So in encryption, if you don't know the secret key, it should be hard for an adversary to discover what the message corresponds to. The adversary sees what's called the cipher text, which is the encrypted text.

The adversary does not know the secret key. If he or she knows the secret key, game over. But assuming that adversary does not know the secret key, it should be computational hard to discover the message, right? That makes sense.

So there's clearly a relationship between this hardness and NP-complete problems, which are computationally hard. And it's not surprising that people have tried to build what are called cryptosystems, which are essentially cryptographic techniques, based on the hardness of NP-complete problems, including graph coloring and knapsack and so on and so forth. But it turns out there's a real subtle difference between the kinds of problems that have been useful in terms of building secure cryptosystems like RSA and NP-complete problems.

And we'll talk about that at the end of lecture. We'll probably spend a bunch of time on that. So let's get started with the symmetric key encryption, which is, at some level, kind of boring from a mathematical standpoint. So we want to spend a lot of time on it.

It's definitely very useful. It essentially assumes that there's a secret key k. And you can think of this as a 128-bit number. Some people want it to be larger, 256.

But it's suddenly at 64. At this point, it's probably not enough, even though 2 raised to 64 is still a fairly large number. With parallelism and with fast computers, it's a little worrisome that the adversary only requires 2 raised to 64 work to enumerate all possible secret keys of 64-bits right? So 122 raised to 128 is much better.

And this is shared between Alice and Bob. So we'll think about the protagonists here as being Alice and Bob. And they want to exchange information. And typically, the adversary is mal for malicious. But there's other ways you can obviously name adversaries.

So the basic equation here is really straightforward. It's C, which is the cipher text. So a little terminology here, cipher text. m is the plain text or the message.

And plain text means you can just read it. Cipher text means it's scrambled. K is, of course, the secret key that's up here. And the e is the encryption function.

Now, in symmetric key cryptography, basically the requirement is that you have to be able to get back the plain text from the cipher text given a public decryption function and knowledge of the secret key. And this should be a straightforward operation, right? So by the way, e of k m, this is a polytime computation.

It's not constant time. It's typically linear time. And you don't really want it to be quadratic time even, because you want to do this vary fast, streaming. You need to send streams of messages, many gigabytes potentially, that are all encrypted.

And this is actually what happens when you get stuff from your satellite and you're downloading movies. That's exactly what happens. It's symmetric key encryption of a lot of data.

So backwards would be dk c. And the only difference here is that everything else stays the same. This is our decryption function. All right.

So that's symmetric encryption is. And there's a requirement of reversibility here. So it's a lot different from one way hashes that we talked about last time. Because here, while you want going from c to m to be hard, it's only hard if the adversary doesn't know k.

If anybody knows k, it should be easy. In fact, that e and d are going to be virtually identical in terms of complexity and sometimes implementation. It's just you run it in reverse, and you get back what you encrypted. That's the way you want to think about it.

So really what happens is that you need reversible operations in order to build e or d, which are the encryption and the decryption functions. So permutation, for example, is reversible. You can always take something and permute it, and you can go backwards.

So you could do something like that. And the permutation, the reverse, looks like that. What is this supposed to be? It's the fact that I have 3 bits here. And they turn into 3 bits over there.

And obviously, if I reverse the permutation and if I just add a simple thing where it was abc and I'm going to convert it to cba, then cba can go back to abc through the reverse permutation. So clearly, this is a reversible operation. But there are other operations that are reversible as well.

Plus can be reversed by a negation. And exclusive OR is simply exclusive OR. Because if you do A exclusive OR B, then you get C. And imagine if you did B again to that, you get A back.

That's what I mean by reversal. Because you have, essentially, A exclusive OR A cancels out. And so it's something like a exclusive OR B exclusive OR B again would give you A.

So if you go look at AES, for example, which is the Advanced Encryption Standard, is really only about four lines of code, maybe it's eight lines of code. But this is a well used, it has been around for a while, symmetric key cipher that runs in 128-bit mode as well as 256-bit mode.

And it's, like I said, a few lines of code. And you'll see operations like this, permutations. And

you'll see hat symbols if you see a C program, which is exclusive OR.

And you can see the encryption and the decryption are identical in implementation. You're just going to run it one way, and you run it again. And you get back the result. Because these permutations are reversible. And the hats are reversible.

And they're all signed integers, so you're just sort of adding them up. And it's just complement arithmetic, so that looks the same as well. So I encourage you to go take a look at AES. I'm not going to spend more time on it.

The key idea here is of symmetry and reversibility. We're going to move away from that. Clearly, we didn't talk about that when we talked about one way hash functions, et cetera. That was a different situation where we had no secrecy. But here, we wanted symmetry.

The big question that you would ask and you should ask yourself when you see this, you say, OK great. I can build symmetric key encryption algorithms. They're actually, I would say, somewhat easier to do than to build hash functions, which have a whole lot of more interesting properties and hard to obtain properties like collusion resistance. But the question really is how do Alice and Bob share the secret key, k?

So you need that. You need this 128-bit number in order for there to be a channel, a secure channel, that Alice can communicate to Bob with and vice verse. And so now you could say Alice sends Bob a letter. But you know, mal could intercept that letter.

And even worse, what wants to do is look at the letter, actually deliver the letter, which is the best thing for him, and Alice and Bob think that they have a secure channel. That's sort of best case scenario for mal, right? So you could have mal in the middle here.

And you've got to worry about stuff like that. So key exchange, let's move on to talk about key exchange. And how does the secret key k get shared? You can't put this out on a website, right? So it has to be the case, sharing is something that has to be secure in the sense that there can't be any eavesdroppers.

So here's my favorite example of a puzzle that most of you have probably heard about. But those of you who haven't, it's really pretty cool. And those of you who have, it's still pretty cool and worth recalling.

And there's actually something that you probably haven't thought about very much even if

you've heard about this puzzle. That's the mathematical assumption made in the solution of this puzzle. That will be interesting to you.

But the puzzle is a pirate puzzle. So you've got Alice and Bob. Let's call it the Caribbean, because that's my favorite ocean. And Alice and Bob are in two different islands.

And we all know there are pirates in the Caribbean, right? And so Alice and Bob want to communicate with each other. And what Alice has are a bunch of boxes and locks and keys.

She's got the keys for her locks and nothing else and the same thing with Bob. So Bob has boxes, locks, keys for his locks that he can put on his boxes. And in this case, Alice wants to send a message to Bob.

And Alice wants to exchange a key with Bob, so they can eventually communicate in a secure way regardless of the pirates or whoever else is listening. So the problem here, of course, is that if you just send a message on a boat-- so the pirates are kind of nice, in a way, that they will deliver messages. But they are curious, right?

So they're very curious. And they will open up boxes. And so that's supposed to be a boat, by the way, not a box, a little mast there.

So I'm not a sailor. But you have these boxes that are going to get delivered. And the deal is this.

If there's an open box, the pirate's will open the box. And if there's any message in it, they might read it, they might throw it away. So clearly, a secret key can be exchanged by just putting a box with a message in it if you don't lock it.

They will not be able to open, they'll not touch, and they will deliver a locked box. But if they ever see a key, they'll keep it. If they ever see kind of a key on the boat, they'll just grab it and keep it. And then the next time around, they see a locked box, they'll stick the key in and try and open it.

All right. So how do Alice and Bob securely exchange a secret where security is based on this notion of piracy, I guess, with the pirates having a certain amount of capability in terms of storing keys, and opening up the locks, but they will not touch a locked box? All right. So that's the puzzle.

How many of you have heard of this puzzle before? So all of you keep quiet. Someone who hasn't heard of this puzzle before, think for a few seconds here and see if a solution occurs to you with respect to going back and forth-- so that's the hint, going back and forth-- and being able to securely exchange a message, which could be a 128-bit secret key written on a little note between Alice and Bob. Yeah? Back there.

AUDIENCE: Are they allowed to pass locks such that--

SRINIVAS DEVADAS: So if you see it locked, they'll throw away the lock, too. If the lock is on the box, then sure, they will deliver that box. But if the key is outside of that box, then they'll keep the key. No-- this is a difficult puzzle. Yeah?

AUDIENCE: Alice could lock the box and send it to Bob.

SRINIVAS DEVADAS: Yeah.

AUDIENCE: And then Bob could also lock the box.

SRINIVAS DEVADAS: Yeah. And then? At that point-- no, you're on the right track. Keep going. Just push it a little more. Couple of more boat rides and we're done. Yeah.

AUDIENCE: And then if Bob sends--

SRINIVAS DEVADAS: So there's now two locks on the box.

AUDIENCE: Yes.

SRINIVAS DEVADAS: And Bob has two locks on the box. And so what's Bob going to do next? What's the logical thing for Bob to do next?

AUDIENCE: Send it back to Alice.

SRINIVAS DEVADAS: Send it back Alice. And now the key's inside. Alice now sees two locks on the box. And one of the locks is Bobs. And other is?

AUDIENCE: Hers.

SRINIVAS Is hers.

DEVADAS:

AUDIENCE: She can unlock it.

SRINIVAS DEVADAS: She can unlock the box and send it back to Bob. And all of this time, the only thing that's gone in transit is a locked box. No keys have gone into in transit, whether they're inside the box or wherever. But the only thing that's moved is a locked box.

So that's good. You're exactly right. So that gets you a Frisbee. Whoops, sorry. We need a secure exchange there. Yeah, that's good.

So that is exactly right. So just to recap, Alice locks box with KA, and that's the key for the lock, sends it to Bob. Bob locks box with KB, sends it to Alice. And Alice unlocks-- oh, I should've said that Alice puts message in box.

And that message has the secret key inside of it. Alice unlocks KA and sends box to Bob. And then Bob unlocks KB and reads message. So that's good. That's all good.

Let's look at it a little more deeply and think about it from a mathematical standpoint, not a physical standpoint. You could think about it from a physical standpoint as well. What is the relationship between this locking, this sequence, in this case a pair of locks, that we require here in order for this to physically make sense? s

I mean, there's different ways that you could add two locks to a box. There's many different ways. One way is to have a box that is locked if I look it over here.

And it doesn't open, because the lid doesn't open. I've got suitcases like that. And then there could be another spot here that looks it as well. So that could be one way. Well, what's another way of having two locks? Yeah

AUDIENCE: Putting a box within a box.

SRINIVAS DEVADAS: Yeah, putting a box within a box. That's great. So if you put a box within a box, then does this work? It doesn't work, right?

So nested locks actually don't work. Because what happened here is that KA was put first, then KB. And if, in fact, you had KA, and then KB out there, then you can't remove KA without removing KB. And this would be the case where you have nested locks.

So the mathematical operation that we require is commutativity between the locks. And the locks need to commute. I want to put KA in first. Like I described, the physical realization could simply be a suitcase with two different positions for the two locks. And any one of those positions locks the suitcase.

So you put KA here, KB here. And then you can take KA out, right? And it's still locked, because you have KB. So this commutativity between the locks and essentially the keys is what's required.

And so now, let's move away from pirates and go into the cryptography domain, pure mathematical domain, and see how this turns into what's called the Diffie-Hellman key exchange, which is a key exchange algorithm or a protocol that under certain conditions give you exactly what you see here. It gives you a secure key exchange. And there's one issue associated with it, and that'll be kind of clear once we write it out here, that we'll get back to.

But Diffie-Hellman key exchange assumes that you have commutative locks. And this is how it works. You'll see what commutes when I give you the equations associated with Diffie-Hellman key exchange. And this is also described in the '70s.

So what we're going to do is we're going to work in a finite field Fp*. And the finite field means that we're going to be doing mod p, where p is prime. And the star means we're going to be only looking at invertible elements only. So we drop the nonvertible elements.

These things aren't particularly important. And so we'll drop 0. And we'll be looking at 1, 2, to p minus 1. So all the numbers that you see are going to be 1 through p minus 1.

Now, what is the analog all this protocol that Alice and Bob, in our pirate puzzle, operated on or ran? What is the analog in the mathematical domain or in the finite field domain? Well, here's what happens.

Alice is going to select a random a. And we're going to assume the g is public. So she just shouts that out to-- Alice can see Bob from her house.

So she shouts out g and shouts out p. They're all public. She doesn't care if the pirates can hear this.

And Alice is going to select a, which is random, and compute g of a. And this is in the finite field, capital G. So you're going to do your mods. And she's just going to send over g of a over

to Bob.

Now, what Bob does is select b and computes g raise to b and sends that over. So g raised to a is being sent over. g raised to b is being sent over, sent over to Alice. So Alice gets g raised to b.

And the key realization here is that Alice can compute g raised to b raised to a, because she knows a. This is all going to be mod p. And we're going to call that K. And thanks to the fact that exponentiation commutes, Bob computes g raised to a raised to b, because Bob knows b, which is also exactly K-- I should say mod p over here. Everything is mod p.

So that's it. That's Diffie-Hellman key exchange. You have now created a shared secret based on the commutativity of exponentiation. And the part that's still missing here with respect to the analogy is the fact that g of a is essentially the locked box.

So what g of a is hiding is a. Because you want a to be hidden here and the same thing with g of b. It needs to hide b. So the problem that the pirates had was they couldn't open up the box.

The problem that the adversary, let's call them mal, has is that he has to invert g of a in order to discover a. And in this particular finite field, and many such finite fields, you can think of this as being what's called a discrete logarithm problem. So if you don't know how to computer logarithms in that continuous domain. And there's tables. And it's pretty easy to do.

But this is what's called a discrete logarithm problem, because we're in a finite field. And we obviously want integers. a is an integer. So we need to discover what does that integer is. Because we're doing the mod p, et cetera, and p is typically on a large number, it's actually a hard problem computationally to do a discrete log.

So when you see g of a, you know g, you know p, but trying to figure out what produced that g of a is a difficult problem. And people have looked at this for 30, 40 years and there's not great algorithms to solve this problem, certainly not anything that's polynomial time solvable. They're all kind of subexponential. And you can make the numbers large enough such that g of a is secure in the sense that it doesn't give away what a is.

So that's the insight here that Diffie and Hellman had, which comes down to the discrete log problem is hard. And what this simply means is given g of a, the discrete log problem is compute a. And the same thing for b, of course.

There's one other thing that you want to say to be precise to sort of just cover the spectrum with respect to how this could break. And that is what's called the Diffie-Hellman problem, for want of a better names and since these are the folks who first came up with it. And the Diffie-Hellman problem is simply that given g of a and g of b, which is what the pirates see and what the adversary mal sees, we should not be able to compute g of a times b, which is exactly what we get here.

g of a raised to b is g of a times b. So given those two things, if there's a way of computing g of a times b, even though you potentially haven't discovered a and b precisely, g of a times b is just the secret key that Alice and Bob exchanged. So you're host. Alice and Bob are host if mal can do this.

So there's two things going on here. You want the Diffie-Hellman problem to be hard. And you want the discrete log problem to be hard. OK So generally, this is how cryptography works.

You set up protocols. And there's some information that's bound to be exposed. You want this information to be hard to reverse to get the crucial information. That requires the computational hardness assumption like the two that we've made here. And then you're off and running.

Your system will break if your computational hardness assumptions are incorrect. And they may be correct for a particular time, for example, the 1970s for particular parameters. But they may end up being incorrect assumptions, at least for those parameters, at a later point of time, simply because computers got faster.

It's like 2 raised to 40 was this huge number when I was your age. Now, it's like nothing. So that's basically part of the game. But the good systems are those that scale where you increase the parameter size and the system and the protocol stays the same. And so you just increase p, for example, in this case. And the discrete long problem is still hard for modern computers.

So those are the good protocols and the good cryptosystems that stand the test of time, not necessarily the ones that have exactly particular parameters. That's hard to do. Because as I said, Moore's law and computers have been getting really exponentially faster. All right.

So there's one main problem with the Diffie-Hellman protocol and the solution to our pirate puzzle. And so can someone tell me-- and it could be just for the sake of the Diffie-Hellman

problem or just in the context of the Diffie-Hellman problem, but also in the context of the pirate puzzle-- what assumption are we making here that's as yet unstated with respect to this secure key exchange being actually secure? Someone. Yeah.

AUDIENCE: If I were to intercept a message from [INAUDIBLE] something of their own back?

SRINIVAS DEVADAS: What does that mean? The pirates see a locked box. So the first step of the protocol, Alice is sending a message inside a locked box with a single lock in it.

You're kind of on the right track. And what could the pirates do in order to break this protocol? We've kind of made an assumption here. And I might have said it explicitly. Yeah, go ahead.

AUDIENCE: [INAUDIBLE] throw the box away.

SRINIVAS DEVADAS: They could just throw the box away. But that doesn't break the security of the protocol. That breaks the functionality of the protocol, right? Yeah, go ahead.

AUDIENCE: Put their old lock on [INAUDIBLE].

SRINIVAS DEVADAS: Ah, that's exactly right, put their own lock on it. You know, if they had locks-- so these pirates don't have locks, right? We're making that assumption.

If they had their own lock with a key, if they just had a lock that locks and they didn't have the key for it, they will wouldn't be able to break the security of the protocol. But if they had a lock and a key for that lock, then they can pretend to have delivered this to Bob. And there's no authenticity here with respect to Alice doesn't quite know whether she's communicating with Bob or not.

She's at the mercy of the pirates to deliver these boxes. So if the pirates had a lock and the key for the lock, then we're in a situation where Alice may have exchanged the key with the pirates. And she thinks she's exchanged it with Bob. And she's, in fact, communicating with the pirates.

So there's a man in the middle attack, which corresponds to the pirates having their own locks and keys. And it's even more trivial in the case of our picture here. Because assuming the pirates know mathematics, they can generate a random number-- and the number could be c, for example-- and just send back g raised to c.

And for all you know, they could actually get Bob to send g raised to b back, but they would

intercept it and replace it with g raised to c. And they know what c is. So what's happening now is so you can [INAUDIBLE]. I won't to go through all of the math here. But you can kind of see it, I hope.

You end up, if you're Alice, exchanging a secret key with the pirates as opposed to Bob. And the way you can set this up is the pirates actually will get into a situation where Alice and Bob think that they're communicating with each other in a secure fashion, but the pirates can listen to all of the messages. They can decrypt all of the messages, because they know what they secret key, k, is. OK.

And remember that the secret key, k, is something that is probably going to be used in a symmetric key encryption scheme eventually to send real messages. So you're going to have ciphered text with that secret key, capital K, over there. And if the pirates or mal get to know what the secret key, k, is through a man in the middle attack, you've got problems. All right?

So the man in the middle attack is something that we have to worry about. What we're going to talk about next is something that addresses this problem. And it may not seem like it's directly addressing the problem. But fundamentally, what's going on here is you need to have authenticity in who you're communicating with.

Alice has to somehow authenticate Bob. And Alice has to know somehow that g raised to b is something that came from Bob. It's not g raised to c that came from somebody else is in the middle. And that's where asymmetric key cryptography and public keys come in, where you have a certified public key associated with yourself.

And maybe you need VeriSign or you need the DMV, or the Registry of Motor Vehicles, RMV, to do this for you. But you create a certified public key, which is associated with your identity. And it's public. You can put it on a website.

And everyone can access it using HTTPS, so they know they're going to the exact website that you've put up. And that gives you a way of identifying yourself. And if you can do that, you can protect against the man in the middle attack using asymmetric key cryptography.

So that's kind of the final part of this puzzle that's associated with authentication and secret key exchange and all of that. Once we do that, you'll know what the functionality is that we require. And then we'll have to talk about how we can build a subsystems.

Cool. Any questions so far? How we doing? OK. So public key encryption, let me just do a little

set up. I said some of this last time. But to make sure we're on the same page, what we have here is we really want a message plus a public key.

And you want to obtain ciphered text using this operation. And this plus is not arithmetic addition. It's just that we're putting these two things together into an algorithm, a public key encryption algorithm, that produces ciphered text. All right.

And this public key, just to reiterate what I just said, is going to be if Alice is producing the message and Bob is getting the ciphered text, this is going to be Bob's public key. And the fact that it's Bob's public key is something that Alice should be able to authenticate using VeriSign, using the Register of Motor Vehicles, what have you. That's what's going to protect against the man in the middle attack. OK.

We're not going to talk a lot about how you can get a certificate. Your MIT certificate is something which corresponds to your MIT ID. It's got information, what year you are, what your name is. And when you generate that certificate, you are getting a certificate of authenticity that you are you.

And of course, you give that away and you hand it to someone else, someone can pretend to be you as well. But that's what's happening when we talk about public keys and you owning public keys. We're not going to, as I said, get into that very much more. I'm more interested in describing this algorithm for public key encryption.

We'll look at a couple that produces a ciphered text given a message and a public key. Now, of course, what Bob needs to do is to take the ciphered text. And this is what Bob's doing. And Bob has a private key that is distinct from the public key and needs to get back exactly the message using a decryption algorithm that corresponds to the message that Alice sent.

OK. And this whole thing is going to work provided knowing the public key, let's call it PK, and the private key. We can't call it PK as well, obviously. So we call it SK.

Knowing the PK does not reveal anything in a mathematical sense about SK. But obviously, in order for this whole thing to work, PK and SK have to have some mathematical relationship. And the different cryptosystems including RSA, and we look at a knapsack cryptosystem, all have different algorithms for encryption and decryption. And they have different mathematical relationships between PK and SK.

And for each of these relationships, you have to show that the adversary has to solve a computationally hard problem in order to discover SK given PK. And it turns out that for most of these systems, it's symmetric in the sense that these algorithms, at least for RSA, you could use either one of these interchangeably. And there's issues associated with that. So we really won't go too deep into that.

But what I said you should hold, which is you have one, it shouldn't tell you anything about the other. There has to be a computationally hard problem associated with discovering one of these only given the one other. And we'll talk about what those hardness assumptions are certainly for RSA and also for another cryptosystem, a knapsack.

So we're going to present RSA, which is this real magical algorithm. It's amazing it works. Every time I prepare for this lecture, I got to relearn somethings. And that's because there's one subtle aspect of this.

It's all about number theory. Number theory can get pretty subtle. But it's also intricate enough that I forget the details.

So let's get started on that. Basically, RSA is based on primes and factoring numbers into primes and using number theory to make sure that you can actually accomplish what this is trying to do. The functionality of RSA should be distinct from the security of RSA. When we talk about the functionality of RSA, we are saying for any message, if Alice uses Bob's public key to encrypt it, the ciphered text resulting from that should be decryptable into exactly the message that Alice sent given Bob's private key.

That's the functional requirement of a public key encryption algorithm or a public key cryptosystem. The security requirement of a public key cryptosystem is what I wrote up there. It's the knowledge of SK should be hidden even given the knowledge of PK. And there's precise computational hardness assumptions that are associated with each cryptosystem.

So let's separate out functionality from security. We're going to talk about functionality for the next few minutes. Alice is going to pick two large secret primes, p and q. So what I'm going to describe here are Alice generating her public key and her private key.

She's going to then publish her public key and keep her private key secret. Bob does the same thing. And then after that, they have to register. And this is not something we'll spend time on beyond me saying it one more time. They have to register their public keys with the VeriSign

or the RMVs like I talked about.

So everyone knows that Alice's public key is this long number. But no one knows Alice's private key. So Alice picks two large secret primes. So these are actually going to result in the creation of our private key.

And then Alice computes N equals pq. So she just multiplies those out. She chooses an encryption exponent, e, which satisfies this little equation, which says that it's relatively prime in relation to p minus 1 times q minus 1. And she knows p and q. So she can compute p minus 1 times q minus 1.

So the gcd of e and p minus 1, q minus 1 is 1. And you can certainly accomplish this simply by choosing e to be a prime. Because then a gcd of a prime with anything else is 1. And it turns out that RSA uses-- this is all going to be public, by the way. e is going to be public. So you can just fix that.

And most RSA algorithms just fix that to be a small number. The encryption exponent is a small number. And the reason it's a small number is because you're worried about performance. And we're going to exponentiate using e. And the smaller it is, the faster the encryption is going to go.

So if you want to encrypt fast and decrypt more slowly, unfortunately, that's the trade off here. You would pick a small e. And then we're going to compute our decryption exponent, which obviously is going to have to be private.

Because that's part of our private key. But that's going to be bigger if e is small. And that's just a trade off.

It's symmetric in the sense that while it's an asymmetric algorithm, it's kind of symmetric in the mathematical sense that the private keys and the public key operations are symmetric. So what is Alice's public key? Well, Alice's public key, which she can then publish, is simply m, e. OK.

Now, the fun starts. We have to figure out what the private key is going to correspond to. And it turns out-- and this is one of those things where how did they ever think of this? And that's still true 40 years later. You get the decryption exponent using the extended Euclidean algorithm.

And this is done by Alice secretly, where what you want is to have the relationship that e times d is 1. And this is mod p minus 1, q minus 1. And there's algorithms out there that would find the inverse that corresponds to d for e or vice versa. And they're polytime algorithms.

As long as you know this number here, mod p minus 1, q minus 1, and you know that Alice knows that, you can get your decryption exponent. And typically, if a is small, as I said, d is going to be large. By the way, the numbers here, p and q, are going to be the roughly 1,000 bits long. So that's essentially-- we're talking about huge primes here. And so n would be 2048 bits in that case.

So the private key, Alice's private key, you can think of as d, p, q. So now, it's clear as to what's public and what's private. n and e are public. d, p, and q are private.

So that's the set up for RSA. And it's not at all clear that RSA accomplishes either of the two things that we need, the first of which is functionality, the fact that encrypting, and then decrypting a message should get you back that message. So that's the first thing we need to look at. And the security part is actually a little bit easier. Because you can see we're going to have to make assumptions about factoring primes and so on and so forth.

Right here, you can just see that immediately. The biggest assumption made by RSA from a computational hardness standpoint is simply that if the adversary sees n, that they should not be able to factor it into p and q. Because if they can do that, it's over. So that's actually easier than the functionality argument.

So why does this work? And amazingly, we can actually do this in about 10 minutes. I'm going to explain to you why this works in 10 minutes. And the only theorem that we'll require on top of this, which I will not approve, because Fermat proved it centuries ago, is Fermat's Little Theorem that says that when you have p being a prime-- you can think of this as a special case.

You take m, and m is an arbitrary number. And if p is a prime, then this relationship holds. So you raise it to the p minus 1 power, and you get 1, mod p. So that's Fermat's Little Theorem that's going to be required. And that's pretty much the only thing that you have to invoke beyond sort of standard mod arithmetic to show that RSA works.

So what's going on here? Let's call phi p minus 1 times q minus 1. Obviously, that's showed up a couple of times. And you may as well represent it by using a smaller, simpler symbol. So

we'll call that phi.

And we are going to say that d equals 1. Mod phi is given to us. And therefore, we can say that ed equals 1 plus k phi. So that's it.

All I'm saying is the remainder. Then you took to the mod with respect to phi was 1. So the actual number was 1 plus k times phi. k is some integer. Think of it as a positive integer.

Remember that we now have the p and the q are over there. And p and q are primes. So p and q are primes. And given that that's the case, we really have two cases to analyze.

Oh, I'm sorry. I missed one crucial point, which I should have told you, which is I gave you this. But I didn't actually tell you what is going on. I mentioned it in passing, exponentiation. But I didn't tell you exactly what the encryption algorithm was and the decryption algorithm was.

And obviously, you need that in order to prove their correctness. I mean, it'd be wonderful if you could prove correctness of this. There exists an algorithm that is such that RSA works or public key encryption works.

So it turns out it's extremely straightforward. c equals m raised to e. And that's part of the usefulness and the power of RSA, which is you take m and you exponentiate it. And you take c and you exponentiate it. And the first is the encryption.

You get the c through encryption, as you can see over there, the ciphered text. And m is the plain text. And you get that back.

So our goal here is to show that you have essentially something where when you exponentiate m raised to ed, it should give you m. For these choices that we have, of e and d, we've set up the d in such a way that m raised to ed-- because if you just go do encryption followed by decryption, you are doubly exponentiating. That make sense?

Ask me questions if this doesn't make sense. This is important. m raised to ed should give you back m. And if you can show that for any m, you're done. That's the functionality of RSA. All right.

So that's encryption and decryption. And so now let's go back to here. Now, ed equals 1 mod phi. Because I've set that up.

This is how I discovered d given e. So that's a given to me. That's part of what's called the key

generation phase of RSA. And that's the mathematical relationship that I keep harping on in terms of the relationship between the public and the private key.

So given that p and q are primes, I have two cases. The first case is that gcd of m, p is exactly 1, which means that the message m. So what I'm comparing here, the two cases, is I have the message. And I'm going to break up the messages into two different categories.

That's it. There are all kinds of messages that are possible. These are arbitrary numbers. I'm going to break them up into two categories, one of which where the message is relatively prime in relation to the prime, p.

And it's not a multiple of p. That's the way you want to think about it. Obviously, gcdmp would be 2 if M were 2p. So the case I'm looking at is gcd of mp equals 1.

And then the another case is going to be trivial, actually, which is gcd of mp equals p. Did I say 2 when I said gcd of 2p, p equals 2? Come on. Wake up. Wow, that's it.

Perfect. Wake up. OK. So gcd of 2p and p is p. So those are the two cases. All right. So by Fermat's Little Theorem, this is really Fermat's theorem. And because he had a last theorem, for some reason, some people call this the Little Theorem.

But it's Fermat's theorem. And we know what that is. I just wrote that out there.

You can now write something that says what I'm going to do is I'm going to just take m raised to p minus 1, which is 1. So this thing is 1. And then I'm going to raise it to k times q minus 1.

And you'll see why I'm doing this in a second. Because I want to get the 1 plus k phi factor here. So I'm taking 1 and I'm raising it to this power, which obviously is going to give me 1 back. So all of that is straightforward.

And then I'm going to multiply this by m. OK. And this is clearly the same as m mod p. Because all I've done is multiply it by 1. All right. So why did I do this?

Well, I did this, because I want to group together these two exponents. And since I've run out of room here, let me just erase and finish this properly. The other case is easy anyway.

And so I can write this as 1 plus k p minus 1 q minus 1. And that, of course, is exactly m raised to ed, right? And so what I've done here is, because 1 times m is clearly m, but if I look at this, this is m raised to ed. So that's clearly m.

That's it. Just figured out that when I have k phi here, that's going to turn into 1, basically, when you exponentiate it. So that's the hard part, actually, as it turns out, of proving RSA's correctness, just introducing this 1 raised to something. And then the easier part is simply the case where the m is actually a multiple of p.

So you have a gcd m, comma p equals p. And in this case, you know that m mod p is actually 0. Because m is a multiple of p. So you're basically exponentiating 0.

What do you do with 0? You're going to get 0 on both sides. So you're sending a message that's essentially 0 mod p And when you decrypt it on that side, you get 0.

But one last line here is simply that m raised to ed is m trivially all of this mod p if m is 0, right? So that was the easy case. So it was really pretty straightforward.

In one case, we took 1 and we exponentiated it and showed the result in a couple steps. In the other case, you had messages that were 0 mod p. So it's very pretty. It works.

And so what happened here? I said that m raised to ed equals m. And what I did here, of course, was I did everything. So it's not quite done, a little slight of hand here as I switched over and talked about mod p.

So I had mod p over here. And I said p and q were primes. And I looked at p first. But what I need to do, just to finish this off, let me do this over here, is I have to do the same argument for q. And I'm going to put it together for n.

And the reason for that is simply that I have n over here. So remember n equals p times q. The encryption and that decryption are going to be done mod o. Certainly, the encryption has to be done mod n, because n is only public number that you have that you can mod with, correct?

So what I've done here, this analysis is for p, can do the same for q. It's exactly the same, because p is a prime and q is also a prime. But I have to just do one last thing, which is put these two things together and say that n equals p times q, so the math is all going to work out.

Let me just write that out. It's not too difficult to explain, once I have this up here. So in both cases of p and q, so when I say both cases, I mean the pk's and qk's. I have m raised to ed equals m mod p. And m raised to ed is the same as m mod q.

And since p and q are distinct primes, we can say that m raised to ed equals m mod N. And that essentially says c raised to d, if you really want to put it all together, which is m raised to e raised to d, equals m mod N, which of course is what we want here. And this thing was also mod N. This is mod N, mod N, mod N.

All right. So that's RSA. That's your first public key algorithm, the first public key algorithm, at least that stood the test of time, still in use today. From a standpoint of computation, this is the hardest part. You have to exponentiate, and you have these large numbers.

And as the years have rolled by, RSA, as I've said, withstood the test of time. But the parameters have increased. Way back then in the '70s, they were thinking about 512-bit primes. In fact, I can't recall whether n was 512-bits or p and q were 512-bits. But if p and q were 512-bits, then n would be 1024.

And now, NSA recommends 8192-bits for n. So there's been an increase. But the nice thing is that it's not like there's an exponential increase in the computation. Because the computation is polynomially related to the number of bits.

So if you double it, I think, if I recall correctly, decryption is going to be the cube of that. Or actually, verifying signatures is probably the cube of that. But don't worry too much about that. The bottom line is that as you double the size of the exponent, you're going to have a fairly small increase in the time required to decrypt or to verify a signature, et cetera. But it has grown from 512 or 1024 to 8192.

And so hopefully, you all understand how RSA works to some extent. I just will leave it at the hardness assumptions here are, like in the case of Diffie-Hellman, two-fold. And the first one is kind of immediately obvious.

Just like it was the case with Diffie-Hellman, where you had g raised to a flying about and obviously that has to hide a, here you got N, capital N, being published. And if anybody could take N and factor it, there may be multiple factorizations. But you're going to get a unique prime factorization. So that's what you want, that unique prime factorization of N.

And if you get that, then you've broken the System because you know what p and q are. And so this is all public in the sense that this algorithm is public. If you're using RSA, this is what you're following.

So the person is trying to figure out what the two primes are that together get multiplied to form capital N. And so that's a factorization problem. And so RSA hardness assumptions are given N, hard to factor into p, comma q. And this is factoring. And then one other thing, which is given e-- so you're not actually breaking the entire cryptosystem. But you're breaking the privacy associated with a particular message.

And so you could break the privacy associated with a particular message. You're given e, because that's public. And you don't know what p and q are. But you know that e is relatively prime with respect to p minus 1, q minus 1, because that's RSA algorithm. And that's a publicly known.

And you also know c, which is the ciphered text. And so what you're doing is you're trying to discover m. So you're trying to break a particular encryption that was created by the RSA algorithm. And you haven't discovered the private key here. That's only discoverable through the factoring problem.

But you could break security if you can find m such that m raised to e is c mod N. So you're doing the searching for an m. So you're trying to discover an m that you can exponentiate to get what you have on the right-hand side.

Because clearly, you can compute what's on the right-hand side. So this is simply called RSA problem. So those are the two computational assumptions that you need to make in order for RSA to be secure. Cool. Any questions?

AUDIENCE: [INAUDIBLE] center?

SRINIVAS DEVADAS: So that's anonymity. Yes. There's ring cryptography. And there's a whole host of protocols. I actually did some of them based on RSA, where you can, by collecting a bunch of private keys together, essentially set it up so it can be verified that the message came from a collection of people, but you can't tell which person the message came from.

So there's just a whole host of things. There's thousands of papers written. There's a wonderful field. I encourage you to look into it if your interests are inclined this way. And it's just gone on and on.

It's become more important with the internet. RSA, the company, was probably founded in the late '70s. And they struggled for a while. And then eventually, they were used for Secure Sockets Layer in Netscape, which was their big break.

And then, of course, Netscape meant the internet was around. And so really, the internet made RSA what it is today. And so just a whole host of wonderful algorithms out there, some of which are based in RSA and some that are broken.

And so let's talk for the last few minutes about all of the fits and starts that occurred in cryptography. And precisely what I'd like to focus on for the time we have left is hardness. So we spent a lot of time talking about hard problems.

And we talked about NP-complete problems that are hard. But they're hard in the worst case. So you have a situation where you have NP-complete problems.

And I'd like to talk a little bit about the relationship between NP-completeness and crypto. Because we've made these assumptions about hardness. Now, what's interesting here is that N composite is clearly in NP, but unknown if NP-complete.

So this is very interesting. The tried and trusted algorithm for public key encryption relies on a computational assumption where the problem associated with that assumption is not even known to be NPC. All right. So that's kind of wild.

So how does this work? Or why does this work? Now, if you take other problems, like, is a graph 3-colorable? And so what does that mean?

Well, you have three colors. And you're not allowed to reuse the same color on two ends of an edge. So if you put red over here, you can put red here, but you can't put red here and there. And so that graph is 3-colorable.

But if you had a click, then this would not be 3-colorable. Because you have all these edges. You have three edges coming out. And so clearly, the degree from a vertex is going to tell you what you have.

So if you have a 4-click over there, immediately it's not 3-colorable. But checking whether a graphic is 3-colorable is NPC. You can use a three set as a way of showing that.

So you can say, oh, wow, maybe I shouldn't be worried about RSA. I should just be building cryptosystems based on 3-colorability. Because it seems like a much simpler problem than all these grungy map that you have out there-- actually, beautiful map that you have out there. OK.

So that's something that's a perfectly reasonable question to ask. And then we have Knapsack. Knapsack is simply you've got a bunch of items and you just want to figure out whether you can get this particular sum S. Knapsack is NPC as well.

And you got a bunch of weights given to you. And the BI are going to have to be 0, 1. So you just want to discover a particular assignment of the BIs, such that you pick the appropriate items to put into the knapsack, right? That's it.

That's a perfectly reasonable problem to potentially use as a basis for computational hardness to go build cryptosystems. And people did that. People did that for years. They tried and they tried.

And they produced cryptosystems, public key cryptosystems, based on Knapsack that look fantastic. And they work from a functionality standpoint in the sense that you would use this Knapsack-- and I'll give you a sense of how this is done in a minute-- problem to encrypt. And then you'd use a different kind of Knapsack problem to decrypt.

And when you encrypt it and decrypt it, you did get that same message back, except the whole world knew what the message was. Because the problem associated with the Knapsack wasn't hard enough. So the computational hardness was what broke the Knapsack schemes.

And then you come down to asking why is it that this problem that's not an NPC seems to have stood the test of time, but all these other problems, like Knapsack and 3-colorability, which is even worse than Knapsack, when people have built cryptosystems based on this, they've all been broken very quickly? And so why is that? What do you think the reason is, sort of at a high level? What does NP-completenss say?

When we talk about complexity, what are we worried about? Most of the time, what are we talking about when we talk about complexity of an algorithm? Or in this case, in the case of a problem, what adjective do we put in front of runtime, for example, then we compute complexity?

AUDIENCE: Worst [INAUDIBLE].

SRINIVAS DEVADAS: Worst case, right? Worst case. In the worst case, you're going to be able to create random graphs where it takes exponential time to discover whether they're 3-colorable or not. But in the average case, all you do is if you have a large graph, if there's one little 4-click in the graph

and you can find it, instantly you know that it's not 3-colorable, right?

So it turns out that's 3-colorability is just the worst thing ever when it comes to cryptography. Because the larger the graph-- and you need this graph to be large. Because anything that's small, is constant time, right? Because so what if it's exponential? It's constant time.

So you need the graph to be large. When you have a random graph that's large, the chances you're going to find a 4-click in a 2,000 vertex graph is pretty high. And so if you just go scan and look for a 4-click, instantly you know that this graph is not 3-colorable, right?

So in the average case, 3-colorability is easy. It's easy to solve in the average case. And the wonderful thing about factoring is as long as the numbers are large, doesn't matter what the numbers are, it's hard to factor in the average case.

So that's the big difference. If you're going to take anything away from the rest of this, it's the difference between problems that cryptographics systems are based on. The systems that have stood the test of time, they're based on problems that are hard on the average.

And the NP-complete problems, like the simple ones here, are hard in the worst case. And this is also true for Knapsack. So that's the essence of it.

I'll just give you a sense. You can read the notes. There's a way of generating secret keys and public keys using Knapsack that I think is kind of interesting that is worth looking at, even though all of these systems are broken.

It's just kind of cool. You know, how would you get encryption out of a knapsack? I mean, you're putting things in a knapsack and taking things out. How can you set it up so you get an asymmetric key system, a public key system, through a Knapsack problem?

So I'll just give you a sense of that. And you can read. I won't finish. But I'd like to do what we did here for RSA in the time that I have left for Knapsack. That is kind of cool.

You get a sense of the variety of different public key cryptosystems that are out there by looking at something that is very different from RSA. So in the Knapsack problem, the general Knapsack problem, what's hard is NPC. There's a super increasing Knapsack problem that's easy, that can be solved in linear time.

What is a super increasing knapsack? Well, a super increasing knapsack is something where I

Wj has this property. So an example of that is 2, 3, 6, 13, 27, 52.

So the rates are super increasing. 2 plus 3 is less than 6. 2 plus 3 plus 6 is less than 13 and so on and so forth. Do you see why super increasing knapsack is easily solvable? I mean, what is a knapsack?

I got a limit on the amount of stuff I can put into the knapsack. And I want to make to be able to say yes or no in terms of whether it fits exactly or not, just in terms of our definition. So what I do here in super increasing knapsack? Yep.

AUDIENCE: [INAUDIBLE] from the biggest to the smallest [INAUDIBLE].

SRINIVAS DEVADAS: Exactly. That means that there's a linear time algorithm that basically solves the problem. And you know that you can do that, and you would get the correct answer. So that's pretty much what you've got. That'll give you the highest weight.

If you have 13 exactly, you know you can't put 52 and 27. You get 13. There's no point in putting 2 and 3 and 6, because that's not going to give you 13. So that's clearly easy.

So we've got an interesting case here, assuming this is all going to work out from an adversarial standpoint, which unfortunately it doesn't, you can look and say, ah, I want encryption to be the super increasing knapsack. Because that should be easy to do. And I want the decryption, not knowing the private key, to be as hard as knapsack. OK.

So that's the kind of thing that you could do if you built a cryptosystem. And people did, Merkle and Hellman. Hellman is the same guy, the second name, in Diffie-Hellman. They proposed this particular system that ended up being broken soon after.

But the idea is that you create a private key. And the private key is a super increasing knapsack. And then you use a private transform in order to get a-- and this is really I put it in quotes, because this was the bug-- "hard" Knapsack problem. And this corresponded to the public key.

And so what you do is you won't actually have to solve Knapsack for encryption, the hard problem. The encryption would just simply take the public key, which is completely public, and you would create the encryption of a message using this particular public key in a polynomial time. But the inversion, not knowing the private key, you would force the adversary to solve what you think was hard general Knapsack problem to actually break the scheme or to get the

decryption.

And so let me show you really quickly how this works with numbers. And we won't have you worry about symbols and things like that. So just give me a couple of extra minutes here.

So let's say that I had a message. Oh, before I do that, let me look at-- let's say, that we chose N equals 31 and M equals 105. This is actually the message.

No, I'm sorry. Capital M is not the message. These are public parameters. And we're going to take-- this is a transform.

Oh, I'm sorry. These are not public. These are private. My bad. So what I'm going to show you is I'm going to take a super increasing Knapsack. And that's exactly what I have up there.

So that corresponds to an easy Knapsack problem. I'm going to convert it using these private parameters, N equals 31 and M equals 105. And so our private key is our super increasing knapsack, which is 2, 3, 6, 13, 27, and 52.

And the public key, what I'm going to do is simply multiply each of these, 2 times N. And I'm going to take mod M. So for each of those values, I multiplied by N, which is 31, and take the mod of 105, and I end up getting 62, 93, 81, 88, 102, and 37.

So you can get a private key and a public key using this private transform. I'll let you look at the notes. But basically what happens is when you take a particular message M, what you end up doing is you want to encrypt it using the public key, which is this quantity over here. And the way you encrypt that is simply by taking a particular message.

And let's say that the message is written as 011000. Then all you do is add up 93 and 81, because those are those two. And you say this is going to get encrypted by 174. So the message encryption is simply a simple operation where you add up weights of the knapsack. So you end up getting 174 out here.

And the hope is, of course, that when adversary sees 174-- and this is the part where things get a little iffy-- that it's hard-- you have to think about lots of numbers here, of course-- for the adversary to figure out that that 174 is actually 93 plus 81. OK. So the diverse is not necessarily an easy problem.

And that's exactly what Knapsack is, right? I tell you what the sum is over there, which is S.

And I tell you what the weights are. And it's hard for you to figure out what the BIs are. So now you see why this didn't work.

So you have to have a situation where in the average case whatever you produce for the ciphered text here, you're sending out the ciphered text according to this cryptosystem, which is 174, and you want to make sure that the adversary can't figure out that this is actually 93 plus 81. Amazingly, people thought they could build systems using this assuming these numbers were much larger than they are here. But that certainly wasn't the case. Because in the average case, you end up being able to break these systems.

The last thing is, of course, you don't want to necessarily solve the hard Knapsack problem associated with this. So what ends up happening is you end up using N equals 31. So if you want to decrypt, you have N equals 31 and M equals 105. And what you're going to do is take this and multiply it by N inverse mod M.

So rather than doing times N mod M, you divide by N mod M. And you can do this operation relatively simply. And you can go back from 174 to figuring out what the actual message was by computing this quantity.

So I'll stop there. I didn't quite get to everything that I wanted to cover. But take a look at the notes. Get a sense for why the difference exists between NP-complete problems and problems that were used in cryptosystems. And happy to stick around and answer questions.