PROFESSOR: So finally, it's our last recitation. And finally, it's my favorite topic-- cryptography-- because I work in this area. So I have probably a little bit more than what's required to tell you. So this recitation is access more primitives, so we'll introduce several more primitives that may be useful to your future work or study. And so the first one is digital signature.

So we have briefly mentioned digital signatures in the lecture, but mainly as an application of Hatch, so now I'm going to introduce it as a standalone primitive. So as you may have already known, digital signature is used for verifying message authenticity. And it's a pair of function sign and verify. So sign takes a secret key and the message. It outputs a signature, which we refer to as sigma.

And verify takes a public key, a message, and a signature, and outputs on either true or false. It either accepts the signature or reject. So we use secret key to sign and public key to verify. That means so if I want to send the message, I should be the only one who was able to sign it.

And everyone can verify that this message indeed comes from me. So what properties do we want from digital signatures? Any thoughts on that? Go ahead.

AUDIENCE: [INAUDIBLE] given the signature in a message [INAUDIBLE] to get the secret key.

PROFESSOR: OK, that's definitely one. I'll put a more general description of what you just said. Any other answers?

AUDIENCE: And only one signature [INAUDIBLE], like on any message coming out, can you have one signature?

PROFESSOR: OK, so what's your name?

AUDIENCE: Hugo.

PROFESSOR: Hugo says a message should only have one signature. Let's think about whether that's necessary. So if my algorithm is a randomized one that for the same message I output many

possible signatures. So why is that bad?

So for any of them, they will all verify if that's how my algorithm works. I think that's fine. That's OK. It's not a bad thing. Actually, randomized signature is considered more secure. They are less efficient. Any other thoughts?

AUDIENCE: Do you care about speed at all, like how long it takes to sign and verify?

PROFESSOR: That's definitely one, but we haven't got any scheme yet, so we care about functionality first. There are faster signatures and slower ones. So the first one is actually very trivial. We first want correctness. What does that mean?

That means if this sigma is indeed generated by the sign function that verified that the output 1. Otherwise, they should put output 0. That's actually the first and the most basic property we want. I don't want to write it because it's-- so the other one.

So your answer is very close that you don't want to extract the secret key. But to make it more general, what we really want is unforgeability. That means if I have the secret key and someone else-- an adversary, who does not have the secret key-- should not be able to sign the message to pretend to be me.

So they should not be able to produce m star sigma star, such that it verifies. Make sense? So what you said is a special case of this. So if they can extract, somehow extract the secret key, then, of course they can forge my signature on any other messages. But we do want to also prevent attack where they cannot extract the secret key, but they somehow can forge another signature.

But usually we want to make the adversary more powerful because then we have higher confidence that we won't be attacked. So adversary is totally reasonable for it to see a bunch of messages from me. Because I am signing messages and output it to the world. So an adversary may have seen some of the message, signature pairs, I generated. But still we do not want to create a forgery.

Now how is that defined? Because see you can definitely send one of these back. That's a valid message signature pair. So our unforgeability requirement is defined to be-- he should not be able to send such a pair where m star is different from any message he has already seen. There is no way to prevent the adversary from sending one of the message signature pairs he has seen before.

So far, pretty straightforward. Now, how can we get digital signatures? So in the early days, researchers-- and it's actually great computer scientists-- they proposed a digital signature can be implemented as the inverse of public key encryption. What does that mean? So I'll use RSA as the example.

So RSA encryption is m to the e mod n. The encryption is c to the d mod n. So the first attempt is we will just use this as our sign function and use this as our verify function. So now the symbol is a little bit confusing. So now I'm signing a message [INAUDIBLE] c.

Let me actually change it. This is RSA encryption. I'm going to transform it into signature scheme where sign signs a message, and verify, raise the signature sigma to the power of e, and checks whether or not I get back my message.

So this actually makes a lot of sense. Why? Because think of m as a ciphertext. Then if I decrypt it, and then re-encrypt it, I should get back my ciphertext. So correctness-- we have correctness.

And why is it unforgeable? Because an attacker does not have the secret key, so he should not be able to decrypt this m here. He cannot run this algorithm. That's the reasoning behind it. So far so good?

But, unfortunately, it is broken. And so I'll give you, say, seven minutes to think about it. Can you come up with an attack, a forgery? You can see a bunch of messages and then output a forgery for a message you haven't seen before. So is the algorithm clear?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Can you speak louder?

AUDIENCE: Is it just the product of any messages?

PROFESSOR: Exactly. So if an adversary had seen a bunch of messages-- because RSA has this sometimes good, sometimes nice, sometimes bad property, that is multiplicative, homomorphic, or to use a less fancy word-- malleable. So if they, an adversary, sees this message, it can set m star to be m1 times m2 and sigma star to sigma 1 times sigma 2. You can check. This is a valid signature, message signature pair.

You take this entire thing raised to d. They are raised to d individually and then multiply

together, and that's exactly this message here. Attack one. OK. There's actually another attack. That's even simpler and tells you this scheme is even more broken.

So all I want to do is to come up with a sigma when it's raised to e, that's equal to m. So I'm going to select the sigma, compute, m sigma raised to e, because e is my public key, mod m. I can do that. And then output sigma m-- oh sorry, m sigma.

I select the signature first, and I raise it to the power of e. I get a very strange message, but it doesn't matter. That's my forgery. OK, so now you can see this scheme is basically totally broken. But they actually come from our, well, several renown scientists.

But why is that a case? Because actually that definition didn't exist when they were trying to when they were working on this problem. And so that definition looks obvious today, but it's actually not obvious at all.

And I think this algorithm came in the 70s, '78. And in '82, Goldwasser and Micali, two professors from MIT, proposed the definition for signature encryption and basically everything in cryptography. And they won another Turing Award for that.

OK, so let's try to fix it. Any ideas? We do not want to change the framework. Let's still use RSA and combine it with some other primitive you have seen to try to fix it.

What do we want to do? We want to break this multiplicative property. And we want to break this, this step, whatever it's called. Go ahead.

AUDIENCE: Can we change the n, maybe?

PROFESSOR: Change this n?

AUDIENCE: Yeah.

PROFESSOR: Right now, just to remind you, it's a product of two primes, OK, pq. It's a product of two primes. It's how RSA works. What's your idea? Go ahead.

AUDIENCE: Before it raised to the power, we can get the hash function of it.

PROFESSOR: Exactly. Let's just make a small change. So sign will be hash of m, raised to d. And verify will be-- just check whether hash of m equals signature is to e. This indeed fixes these attacks. Why? Because now you need-- well, if you do this-- hash of m and 1 times hash of m2 is not

going to be hash of m star because hash is supposed to be [INAUDIBLE] random.

That's not going to work. And here, what the attacker needs to do is to find hash of m, such that it's sigma raised to e. It can still do this, but it does not know what this message is because of the one-wayness of hash function. If we use a good hash function there, then it indeed fixes both the attacks. But we have seen the lecture that this hash function also needs to be collision resistant. Remember that? A question?

AUDIENCE: Isn't the message public?

PROFESSOR: Yeah, the message-- oh, OK. Good point. Oh, no, but, OK, you are talking about this attack, right? So the attacker needs to find the public message, but all he can do is select the sigma, and raise it to e.

That's going to be its hash of m. And then he cannot figure out where this m is. But what about the other way?

AUDIENCE: I mean, if he has two messages, he can still get m star, and then get hash of m star.

PROFESSOR: OK, so he then he gets, has of m1. He gets hash of m2. But you need to find the m star, such that its hash is the multiplication of these two. And, yeah, he does not know how to find that message. So if the hash is not multiplicative, one-way, and collision resistant, then it seems that we have fixed all the attacks we know.

However, how do we know there are no other attacks? So actually, indeed, this is a good idea. We have several national standards that just use this but slightly differently. I can-- this is just for your information. So there's a standard called [? NC, ?] whatever-- X93.1.

It uses RSA, this word padding, so it takes the hash of the message and pad with this hex stream, and prepended and append another hex stream. Why do they do that? They don't know either, but they just think it's probably more secure than only using a hash. There's another standard that has a different steam and a difference stream here, and it doesn't matter. So that's indeed a weakness of these types of approaches.

So their security is what we call ad hoc. We do not know how to break them. But we do not know how to prove they are secure either. Yet, that's what people do in practice.

So, unfortunately, that's all I can tell you today, so how not to construct the digital signature. I

cannot tell you how to construct the secure digital signature because that's out of the scope of this class. And it's a major topic in cryptography. Any questions so far? Go ahead.

AUDIENCE: The hash function here is the one way, yeah?

PROFESSOR: Yes, it's one-way, collision resistance, and--

AUDIENCE: So what is the use of using the RSA? Couldn't we just use the only hash function then?

PROFESSOR: OK, good question. So, OK, let's be clear what you're saying.

AUDIENCE: OK, never mind.

PROFESSOR: Can you answer your own question?

AUDIENCE: So my question was, well, why do we have to use the RSA? Why, when we have the hatch function? You want me-- so [INAUDIBLE] couldn't create the forgery. [INAUDIBLE].

PROFESSOR: How does it create a forgery? Just answer your own question. Let everyone else know. Maybe they have the same question. So answer your own question.

AUDIENCE: So my answer is so adversary can't just choose random message and hash it and [INAUDIBLE].

PROFESSOR: Yeah. What's the problem? Problem is that a hash function is a public function that everybody can compute. So the attacker just chooses a message, compute as hash, so using a hash is not a signature. But good point. I'm actually coming to that. So far we have seen three major primitives-- private key encryption, public key encryption, and digital signature.

So if we categorize them a little bit-- so these two are asymmetric key. They are public key and secret key. This one is symmetric key. And these two are for secrecy. They are trying to hide the message. And this one is for integrity. Meaning, the message is what the sender sends.

So you can see we are missing one primitive here. What if the two parties, they do share a secret key, and one party wants to verify the other party? The other message indeed does come from the other party. So indeed we do have a primitive for that. It's called message authentication code.

So its definition is basically exactly the same as digital signature. I'm just going to change it here. Except that it has only one key. So the sign function is replaced by a MAC. And there's

no notion of secret key and public key. We have only one key. And how do we verify?

OK, so verify function basically just becomes the other guy also recomputes the MAC of the message and checks whether that's the signature. So verifier just to recompute and compare correctness-- we also want correctness. We also want unforgeability And it's defined in exactly the same way.

Now, actually, I would have asked this question here-- is hash a valid MAC? The answer is still no because MAC is a public function that everyone can compute, and it's trivial come up with a forgery. So thank you for asking that question. But the hash is actually very close.

How can we get a message authentication code? So several ideas. Can we just hash the key concatenated with the message? Then some other random attacker who doesn't have the key does not know how to compute this thing. That's a reasonable idea.

But, well, if we can do it this way, how about we do the message concatenated with the key. Or if you want, you can do key concatenated with message and then concatenated with the key. So it turns out this doesn't work for some very advanced reasons. And this one may or may not.

For SHA1, it doesn't work, unfortunately. And for SHA3-- that's the replacement for SHA1 and SHA2-- it actually works. So the simplest MAC we can imagine is just to choose SHA3 as the hash function, and input is the key and then the message. Not the other way. It's also, just FYI, purpose.

By the way, there's another reasonable thought. That is, how about we encrypt the hash? Now, everyone can compute the hash, but they don't know how to encrypt. If I use, say, a secret key encryption, this turns out to be wrong as well.

That's digital signature in MAC. But one caveat here, our unforgeability is defined this way. A little bit strange, but it makes sense. But it indeed has some weakness in some applications.

So imagine, say, I send you a message-- today's recitation is canceled. And it has my signature on it. So you can verify it indeed comes from me. But once I send that message, every of you has that message. So next week, one of you can send that message again, saying, today's recitation is canceled.

Then you have no idea whether it's indeed me sending the message again or someone doing

an April Fool's Day joke. So how do we prevent that? Well, of course, one thing I can do is if I'm smart I'll say, today, like in parenthesis, May the 8th, recitation is canceled.

Then you cannot repeat that message. But we want to protect human stability. That's the whole point of cryptography. So one thing we could do, let's see. Very simple modification.

When I sign the message, I'll sign 1, concatenated with my message. Next time I sign 2, concatenated with my message. And then 3, 4, and just have this counter that keeps incrementing. It naturally fixes. So you can verify if you receive the same message with the same counter, then you know it's someone else who is resending it.

So that's one thing we need to do for signature in practical use. Now, consider another totally different application. So say I think everyone uses Google Drive, Dropbox, something like that. You store a bunch of files on this cloud server. Now you are here.

You'll have a, say, cell phone, and you can access your files. But how do you know when you read a file, it is indeed your file unmodified? How do, maybe Google messes with you, or there's someone in the middle who changes your file?

Usually, most people do not care about that, while in cryptography, we do care about that. So in that case, MAC and signatures do not help us. Why? Because if you just store a MAC alongside each file, what went wrong? Go ahead.

AUDIENCE: You need to modify the MAC too.

PROFESSOR: But if they modify the file, they do not know how to generate a MAC for their version of the file. But what they can do is you have this file and then you come and write it, and you generate a new MAC. When you read it, they give you the old version. That has the valid signature or MAC on it because you generated that for it. You all see the problem? You haven't seen the problem?

AUDIENCE: What do you mean they give you the old version?

PROFESSOR: OK, so you have this file. You generate a MAC, but you-- at some point, you want to update the file. You want to update this file to this file prime, and generate a new MAC. Maybe then file double prime, MAC double prime. In this application, we want freshness.

When you read this file, you want the latest version of the file. So it should be what you wrote

there last time. But when you are trying to read the file, an attacker can give you this pair. If you check the MAC, it's going to match. This is also a valid message MAC pair.

Now, everyone sees the problem. OK, so what can we do? Well, one thing we could do is store all these MACs here on your phone. MAC1, MAC2-- a MAC for every single file. But if you do that, in fact, we do not need MAC anymore. We can just use hash.

So I'll say sigma-- I'll use sigmas, but they mean hashes. This is probably good enough in practice. I'll say these files are x1, x2, x3, x4. Now you just create a hash for each of them and store them locally. And the model here is that an attacker cannot modify files on your own computer or on your own phone.

And then you can download the file. Match-- compares it with the latest version of the hash, and then you're convinced that it's the latest version. This is probably a good enough solution. The only downside is that we do have to store a lot of hashes if you have a lot of files.

Or in our algorithmic terminology, we say your space complexity is o of n. Here, I mean your local space. So can we somehow reduce the local space complexity? Well, one thing we could do is to concatenate all the files together, generate a single hash, and store that one hash.

So hash everything in one try. Then we do have o of 1 space, but there's a bigger problem. Can anyone tell me?

AUDIENCE: You don't know which file to modify?

PROFESSOR: Oh, OK, I think you are thinking in the right thing. So how do I verify? I cannot verify a single file. I have to download all the files and recompute a hash to verify. So the time complexity is o of n.

And, also, if I want to update this file, I have to recompute the hash. That involves, again, downloading all the files and feed them into that hash. And we do have a better solution than both of them, which is called a hash tree or a Merkle tree, which was invented by Merkle.

What we will do is so first for every file, we're going to create a hash. Let me, again, use sigma because h is unclear whether it's a hash value or a hash function. Sigma 2, sigma 3, sigma 4. So I said a hash tree. And guess what's the next step to do?

AUDIENCE: Cross the hashes.

PROFESSOR: Yeah. Exactly. We're going to create a sigma 5, which is the hash of sigma 1, concatenated with sigma 2. So we do the same thing here. And so you all know what it is, right? I don't need to write it. And keep going until we got a root hash.

And now we're going to store this thing locally on the side. So what's the local storage complexity? o of 1-- we're only storing one hash locally. Now, what's the time complexity? OK, so how do I verify?

Yeah. Log in-- how do I verify? I need to, so, first verify if this hash matches, and then read this hash, and verify whether this link matches, and verify whether this one matches, and then I'm done. If I want to update, I also need to update this hash, then it causes this hash to change and then that hash to change.

But it's always some path in that tree. It doesn't affect anything globally. Question.

AUDIENCE: But you're not storing like sigma 5?

PROFESSOR: Say again.

AUDIENCE: You're not storing sigma 5.

PROFESSOR: I am not. I have to go ahead and read it.

AUDIENCE: From where?

PROFESSOR: From Google Drive or Dropbox.

AUDIENCE: So are we sure that that is secure?

PROFESSOR: OK, yeah, so that's the next thing we're going to do. Is this secure? Or in other words, can the adversary change one of the files, and somehow maintain the same root hash? That's your question then. Of course, we assume the hash is collision resistant. Or I should say if the hash is collision resistant, then this hash tree is collision resistant.

Any intuition? Or anyone wants to prove it? Go ahead.

AUDIENCE: So like if the root eventually one of the leaves will be different because it changes.

PROFESSOR: Yep.

AUDIENCE: Now, you want the root to be the same than the other hash has to be different, but there's no collisions. I mean, it's hard to find the other hash.

PROFESSOR: Correct, so I'll just repeat what you said, but I'll start with the leaf because that's easier for me to think about. So say I change this one, this block. Now, I claim this hash here will change. If it doesn't, then I have found the collision.

Because this x4 prime has the same hash as the original x4. So if this sigma 4 changes, then sigma 6 will change. Otherwise, I have found the collision. Because this sigma 3 concatenate with the new sigma 4 is my collision. So same argument-- either this one changes, or I have found the collision. I repeat the argument all the way to the root. Any question about that?

AUDIENCE: What if like the adversary changes like two hash options--for example, x1 and x2-- but sigma 1 and sigma 2 changes, but sigma 5 stays the same?

PROFESSOR: OK, so then we have found the collision that is sigma 1 concatenated with sigma 2. That's a collision with the new sigma 1 concatenated with the new sigma 2. Make sense?

AUDIENCE: If the concatenation stayed the same, like sigma 1 and sigma 2 concatenation.

PROFESSOR: So if the concatenation stayed the same, that means both of them are the same.

AUDIENCE: They had to make sure they are changed?

PROFESSOR: So I'm not sure I understand your question. So concatenation is basically just a bunch of bits then followed by another bunch of bits. If this entire thing is the same, that means this part is the same and this part is the same. And if your sigma 1, new sigma 1 is the same as your old sigma 1, that means I have found a collision here. Because we changed it, but your sigma doesn't change.

So lastly, I'm going to do a quick review of the knapsack problem because I think in the lecture, we may run out of time and I didn't mention everything. So if you recall, the knapsack cryptosystem, it says you have a knapsack problem. I'll call u1 to un. And then we're going to transform it.

OK, this is a super increasing sequence. I'm going to transform into a general one by multiplying n and then mod m. So this is an easy problem, and that is a hard problem. So how do I encrypt? I'm going to take a subset sum, which is mi, Wi, where mi is the i-th bit in the

message.

So how do I decrypt? I'll take this, transform this s back to the super increasing domain by multiplying inverse of n. So that's going to be inverse of n multiplied by this mi Wi. That's how I encrypt it. And then each Wi is n times ui.

So far so good. So that gives me mi times ui sigma. Of course, every step is modulo m. So the first thing I'm going to claim is that m has to be larger than sigma ui. If that's the case, then the t-- my t is just this subset sum.

So if I solve this knapsack problem, I get the same answer as solving the original, the general knapsack problem. If my m is not that large, if the m is too small, then I have a problem. Because then my t will be the subset sum minus sum multiple of m. Then it's a different problem. I do not get the same message back.

OK, then we have a problem. So because we defined density to be n over the log of max ui. Does everyone remember this part? So each ui is in the range of 1 to m, or maybe 0 to m. If I have a bunch of them, then this is not super rigorous.

If I have a bunch of them, chances are that some of them are very close to m. Because it's unlikely that all of them are small. So this thing is roughly n over log of m. So then we have a dilemma. If we set m to be a small number, then my density is fine, but that means all of my ui's needs to be small because m needs to be greater than the sum of them.

If all the ui's are small, then I have a very limited choices of them, then actually an attacker can just guess what ui I chose by a brute force algorithm or something like that. And if I choose m to be large, or if I choose all the ui's to be large, to choose them from large range, then my m is going to be very large. And this density is low. And that's vulnerable to the low density attacks. And so how low a density is considered low?

So several people proposed that based on heuristics, that if this density is less than 0.45, then it's considered low density, and it can be attacked. And this threshold had been improved. So but while most of the knapsack cryptosystems are broken, there are few that have so far stood the test of time. So they are still interesting because knapsack problems, knapsack cryptosystems will be much faster than RSA or any number theory based, because we are just adding numbers here.

An RSA have this operation where m is a 1,000 bit number, and e is also 1,000 bit number.

And take this exponentiation is actually very slow. So knapsack cryptosystem are still interesting. However, the original motivation turned out to be unsuccessful. The original motivation is to base cryptography on the NP complete problem.

That's not going to work because NP problems are hard, only in the worst case. And we need cryptography to be hard in the average case. Because if they are only hard in the worst case, that means there are several instances of this problem that are hard. So either you pick a secret key that doesn't correspond to a hard problem, or you pick a secret key that's corresponds to a hard problem.

But everyone else picks the same secret key because everyone wants to be secure. That's the reason why it's unlikely to get cryptography from NP hard problems. That's all for today's recitation. And thanks everyone for the entire semester. Thank you for participation.