SRINIVAS DEVADAS: All right, let's get started. Good morning everyone. I see a lot of tired faces. I'm not tired. Why are you tired?

[LAUGHTER]

I only lecture half the time. You guys take the class all the time. So today's lecture is about hash functions. And you may think that you know a lot about hash functions, and you probably do.

But what we're going to do today is talk about really a completely different application of hash functions, and a new set of properties that we're going to require of hash functions that I'll elaborate on. And we're going to see a bunch of different applications to things like password protection, checking the integrity of files, auctions, and so on and so forth. So a little bit of a different lecture.

Both today and on Thursday I'm going to be going to be doing cryptography and applications, not too much of algorithms. But we will do a little bit of analysis with respect to whether properties are satisfied, in this case by hash functions or not. So let's just dive right in.

You all know what hash functions are. There's no real change in the definition. But the kinds of hash functions that we're going to be looking at today are quite different from the simple hash functions, like taking a mod with a prime number that we've looked at in the past. And the notion of collisions is going to come up again, except that again we're going to raise the stakes a little bit.

So a hash function maps arbitrary strings-- let me do this right. So you're not making a statement about the length of the string. You will break it up, even if you had a string of length 512, or maybe it was 27, you do want to get a number out of it. In a specific range there's going to be a number of bits associated with our hash functions. And previously we had a number of slots associated with the output of the hash function. But the input could be arbitrary.

And these arbitrary strings of data are going to get mapped, as I just said, to a fixed length output. And we're going to think about this fixed length as being a number of bits today, as opposed to slots in the hash table. Because we really aren't going to be storing a dictionary or a hash table in the applications we're going to look at today. It's simply a question of computing a hash.

And because the fixed length output is going to be something on the order of 160-bits, or 256-bits, there's no way that you could store two arrays to 160 elements in a hash table, or even two arrays to 64 really. And so we're going to just assume that we're computing these hashes and using them for certain applications.

I just wrote output twice I guess. So map it to a fixed length output. We want to do this in a deterministic fashion.

So once we've computed the hash of a particular arbitrary string that is given to us, we want to be able to repeat that process to get the same hash every time. We want to do this in a public fashion. So everything is public. There's no secrecy.

There's keyed hash functions that we won't actually look at today, but maybe in passing I'll mention it next time. We're not looking at keyed hash functions here. There's no secrets in any of the descriptions of algorithms or techniques I'm going to be describing today.

And we want this to be random. We want it to look random. True randomness is going to be impossible to achieve, given our other constraints. But we're going to try and approximate it with pseudo-randomness. But we'd want it to look random, because we are interested-- as we were in the case of dictionaries and the regular application of hash functions-- we are interested in minimizing collisions.

And in fact we're going to raise the stakes really high with respect to collisions. We want it to be impossible for you, or anyone else, to discover collisions. And that's going to be an important property of collision resistance that obviously is going to require randomness. And those are the three things we want, deterministic, public, and random.

And so just from a function description standpoint you have 0, 1 star here, which implies that it's an arbitrary length strength. And we want to go to 0, 1 d. And this is a string of length d. So that means that you're getting d-bits out from your hash function. And here the length is greater than or equal to 0.

So that's it. Not a lot that's new here. But a few things that are going to be a little bit different. And there's some subtleties here that we'll get to.

I want to emphasize two things, one of which I just said. There's no secrecy, no secret keys here in the hash functions that we are describing. All operations are public. So just like you had your hash function, which was k mod p, and p was a prime and p was public and known to everyone who used the dictionary, everything here we are going to be talking about is public. So anyone can compute h.

And we're going to assume that this is poly-time computation-- not too surprising-- but I'm being quite flexible here. When you look at dictionaries, and you think about using dictionaries, and using it to implement efficient algorithms, what is the assumption we kind of implicitly made-- are perhaps explicitly in some cases-- with respect to computing the hash? Anybody? Yeah?

AUDIENCE: Constant time?

SRINIVAS DEVADAS: Constant time. We assumed-- so this is not necessarily order 1, right? So that's important. So we're going to-- I want to make sure you're watching. So you're going to raise the stakes even with respect to the complexity of the hash. And as you'll see, because of the desirable properties, we're going to have to do that. We're going to ask for really a lot with respect to these hash functions.

Nobody can find a collision, right? And if you have something as simple as k mod p, it's going to be trivial to find a collision. And so these order 1 hash functions that you're familiar with aren't going to make the grade with respect to any of the properties that we'll discuss in a few minutes. All right, so remember this is poly-time computation.

And there's lots of examples of these hash functions. And for those of you who are kind of into computer security and cryptography already, you might have heard of examples like MD4 and MD5. These are versions. MD stands for message digest. These were functions that were invented by Professor Rivest.

And they had d equals 128 way back when-- 1992, if I recall-- when they were proposed. And these algorithms have since been broken in the sense that it was conjectured that they had particular properties of collision resistance that it would take exponential time for anybody to find collisions. And it still kind of takes exponential time, but 2 raised to 37 is exponential at one level, but constant in another level. So you can kind of do it in a few seconds now.

So a little bit of history. I'm not going to spend a lot of time on this. MD5 was used to create what was called a secure hash algorithm. This is 160-bits. And this is not quite broken at this point. But that people consider it broken, or soon to be broken.

Right now the recommended algorithm is called SHA-3, secure hash algorithm version three. And there was a contest that ran for like 18 months, or maybe even longer, that eventually was won by what turned into the SHA-3. And they had a different name for it that I can't recall. But it turned into SHA-3.

And what happened along the way, as we went from MD4, MD5, SHA-1 to SHA-3, is that this amount of computation that you had to do increased. And the complexity of operations that you had to do in order to compute the hash of an arbitrary string increased to the point where-- you want to think about this as 100 rounds of computation. And certainly order d computation, where d is the number of bits. And perhaps even more. So it's definitely not order 1.

So as I said a little bit of context with respect to the things that are out there. At the end of the lecture I'll give you a sense for how these hash functions are built. We're not going to spend a lot of time on creating these hash functions. It's really a research topic onto itself and not really in the slope of 6.046. What is in the scope of 6.046, and what I think is more interesting, which is what we'll focus our energy and time on, is the properties of these hash functions. And why these properties are useful in a bunch of different apps.

And so what is it that we want? We want a random oracle. We want to essentially build something that looks like that, deterministic, public, random. And we're going to claim that what we want is this random oracle which has all of these wonderful properties that I'm going to describe. I'm going to describe the random oracle to you, and then I'm going to tell you about what the properties are.

And then unfortunately this is an ideal world and we can't build this in the real world. And so we're going to have to approximate it. And that's where the MD4's and the MD5's and the SHA-1's came in, OK? So this is not achievable in practice.

So what is this oracle? This oracle is on input x, belonging to 0,1 star. So that could be an arbitrary string. If x not in the book-- so there's this the book, all right? And there's this infinite capacity book that has all of the computations that were ever done prior. And they're always stored in the book.

And that's how we're going to get determinism. Because this book initially gets filled in. All of the entries in the book are filled in using pure randomness. So you flip a coin d times to determine h of x.

So that's basically it. And you just keep flipping. You have to flip d times. And so if x was 0, you flip d times, d was 160. You flipped a coin 160 times and got a string. If x were 1, flip 160 times, you get a different string with very high probability, obviously. And so on and so forth.

But what you do is you have this book. So you're going to record x h of x in the book, OK? So at some level your hash function is this giant look-up table in the sky, right? Actually not giant, infinite capacity look-up table in the sky. Because you can put arbitrary strings into this.

And if it's in the book-- this is obviously the important part that gives you determinism-- then you return y, where x and y are in the book, OK? So you get a random answer every time, except as required for consistency with previous answers. So the very first time you see a string, or-- and the whole world can create this book. It's public.

So if I created the book at first with a particular string, let's say Eric. I was the string. And I'm the one who put the entry-- x equals Eric, and h of x, h of Eric equals some random 160-bit string-- into the book, I get credit for that, right? But if you come a nanosecond later and ask for h of Eric, you should get exactly what got put into the book when I asked for h of Eric. And so on and so forth.

So this is true for everybody. So this is like-- I mean basically impossible to get. Because not only can anybody and everybody query, you have to have this ordering associated with people querying the book. And you have to have consistency. All right. So everyone convinced that we can't build this? All right.

If you took anything out of this lecture, that's what you should take. No, no. There's a lot more.

So we want to approximate the random oracle. And we're going to get to that. Obviously we're going to have to do this in poly-space as well.

So what's wrong with this? Of course this picture is I didn't actually say this, but you'd like things to be poly-time in terms of space. You don't want to store an infinite number-- this is worse than poly-time, worse than exponential time, because it's arbitrary strings that we're talking about here, right? So you can't possibly do that. So we have to do something better.

But before I get into how we'd actually build this, and give you a sense of how SHA-1 and MD5 were built-- and that's going to come a little bit later-- I want to spend a lot of time on the what is interesting, which are the desirable properties. Which you can kind of see using the random oracle. So what is cool about the random oracle is that it's a simple algorithm. You can understand it. You can't implement it. But now you can see what wonderful properties it gives you. And these properties are going to be important for our applications, OK?

And so let's get started with a bunch of different properties. And these are all properties that are going to be useful for verification or computer security applications. The first one, it's not ow, it's O, W. It's one-wayness, all right? So one-way, or one-wayness. And it's also called-- you're not going to call it this-- but perhaps this is a more technical term, a more precise term, pre-image resistance.

And so what does this mean? Well this is a very strong requirement. I mean a couple of other ones are also going to be perhaps stronger. But this is a pretty strong requirement which says it's infeasible, given y, which is in the-- it's basically a d-bit vector, to find any x such that h of x equals y. And so this is x is the pre-image of y.

So what does this say? It says that I want to create a hash function such that if I give you a specific-- we call it a 160-bit string, because we're talking SHA-1 here, and that's the hash-- I'm going to have, it's going to have to be impossible for me to discover an x that produced that 160-bit string, OK? Now if you go look at our random oracle, you realize that if you had a 160-bit string, and perhaps you have the entire book and you can read the entire book.

It's an infinite capacity book. It's got a bunch of stuff in it. And know that any time anyone queried the book the first time for a given x, that there was this random 160-bit number that was generated and put into the book. And there's a whole lot of these numbers, right?

So what's going to happen is, you're going to have to look through the entire book, this entire potentially infinite capacity book, in order to figure out if this particular y is in the book or not. And that's going to take a long time to do, potentially, OK? So in the case where you have a random oracle you'd have to go through and find-- looking at the output hash corresponding to each of the entries in the random oracle, you're going to start matching, match, match, match, match, it's going to take you exponential time. Well actually worse than that, given the infinite capacity of the book.

So this clearly gives you that. Now you may not be a completely satisfied with that answer because you say well, you can't implement that. But we'll talk a little bit, as I said, about how you could actually get this. But what's-- I should be clear-- is that the simple hash functions that we've looked at in the past just to build dictionaries do not satisfy this, right?

So suppose I had h of x equals x square mod p. Is this one-way, given a public p? No of course not, right? Because I'm going to be-- it's going to be easy for me to do something. Even though this is discrete arithmetic I could do something like, well, I know that what I have here-- actually let's do it with something that's simpler, and then I'll talk about the x squared.

If I had something as simple as x mod p, I mean that's trivially broken in terms of one-wayness. Because I know that h of x could be viewed as the remainder. So anything-- if this is h of x, and let's just call that y for a second, because that's what we had it out there. Something that's a multiple of y plus the remainder-- so I could have a-- is that right? Is that what I want? Yeah.

No, plus y. So I want a of-- well since I can't figure it out, why can't you? What do I need to put in there in order to discover an x that would produce a y? Can I write an equation? Yeah?

AUDIENCE: Could you just write y itself?

SRINIVAS DEVADAS: Just y itself. That's right. Good point. Just y itself in this case. Good. I knew you guys were smarter than me. This proves it.

So if you just take y-- and y remember is going to be something that's 0 to p minus 1, right? And that's it. It just goes through, right? So that's a trivial example, right?

Now if I put x squared in here, obviously it's not y, but I could start looking at-- what I have here is I'm going to get y that looks like x squared. But I could take the y that I have, take the square root of that, and then start looking for x's that give me the y that I have. Actually it's not a complicated process to try and figure out, through trial and error potentially, what an x is that produces a particular y for the kinds of hash functions that we've looked at, all right?

Now as you complicate this equation it gets harder. Because you have to invert this set of equations. And that's what the game is going to be when you go create one-way hash functions. The amount of computation that you do in order to compute the y is going to increase to the point where, as I mentioned, you have 80, 100 rounds of computation, things getting mixed in.

And the hope is that you create this circuit, if you will, that has all this computation in that. Going forwards is easy, because you've specified the multiplications and the mods and so on and so forth. But not all of these operations have simple inverses. And going backwards, which is what you need to do in order to break one-wayness, or discover the x given a y, is going to be harder and harder as the computations get more complex, OK?

So everyone have a sense of what one-wayness is? So that's one-wayness. There's four other properties, two of which are very related.

CR and TCR. So CR is collision resistance. It's infeasible to find x and x prime such that x not equal to x prime, and h of x equals h of x prime, which is of course a collision. OK? And that just says you have this crazy hash function where you can't discover collisions. Well it would be absolutely wonderful. In fact that's what we wanted when we built dictionaries.

But why don't we use SHA-3 in dictionaries? Why don't we use SHA-3 in dictionaries? Yeah?

AUDIENCE: Because it's more complicated than we need.

SRINIVAS DEVADAS: Yeah, it's horribly slow, right? It would take longer to compute the hash than access the dictionary, when you actually had a reasonable dictionary that maybe had some collisions. I mean you just go off and you have a linked list, you can afford a few collisions, what's the big deal, right? So it just doesn't make any sense to use this level of heavyweight hash function, even if it satisfies collision resistance-- which some of these are conjectured to do-- for the applications we've looked at.

But there'll be other apps where collision resistance is going to be important. So that's collision resistance. And then there's-- TCR is target collision resistance. It's a weaker form-- so sometimes people CR strong collision resistance, and TCR weak occlusion resistance. We'll use CR and TCR here.

And this says it's infeasible, given x-- so there's a specific x that you want to find a collision for, as opposed to just finding a pair that goes once to x and x prime. And any pair would suffice to break the collision resistance property. But TCR says is I'm going to give you a specific x. And I want you to find an x prime who's hash collides with the hash of x, OK? That's TCR.

OK that's TCR for you. And that just to be clear, I think you probably all got this, obviously we want this here because we have a deterministic hash function. And it's a trivial thing to say that if you had x, and you had x again, that you get the same hash back from it. That's a requirement, really. So we want two distinct x and x primes that are not equal that end up colliding. That's really what a collision is.

And so you see the difference between CR and TCR? Yup? Yeah?

AUDIENCE: Are we to assume that given an x it's very easy to get the h of x back?

SRINIVAS DEVADAS: So the question was, given an x, it's poly-time computation to get h of x. Absolutely. Public poly-time computation given an x to get h of x. So going this way is easy. Going this way-- I ran out of room-- hard. OK?

AUDIENCE: So does that mean that TCR is basically the same as 1?

SRINIVAS DEVADAS: No, no, no, absolutely not. TCR says it's OK. You can compute this. You can get x. And you can get h of x. So given x, you know that you can get h of x.

I didn't actually put that in the definition. And maybe I should have. So given x you can always get h of x. Remember that. It's easy to get h of x. So any time I say given x, you can always add it, saying given x and h of x.

So I'm given x. I'm given h of x. I obviously need to map-- I need to discover an x prime such that h of x prime equals h of x, OK? Now you have situations where for-- it may be the case that for particular x's you can actually do this. And that's enough to break TCR. So you have to have this strong property that you really don't want to find collisions are for some-- even if there's a constant fraction of x's that break the TCR property, you don't like your hash function, OK? Because you might end up picking those and go build security applications using those properties.

I want to talk a little bit about the relationship between OW, CR, and TCR. So I'm going to get back to that. And we're going to talking about hash functions that satisfy one property but don't satisfy the other. And I think your question will probably be answered better, OK? Thanks for the question.

So those are the main ones. And really quickly, if you want to spend a lot of time on this-- but I do want to put up-- I think I'll leave these properties up here for the duration. Because it's important for you to see these definitions as we look at the applications where we require these properties, or a subset of these properties.

But that we have pseudo randomness. And this is simply a function of the fact that-- so this is PRF-- we know we can't build a random oracle. And so we're going to have to do something that's pseudo-random. And basically what we're saying here is the behavior is indistinguishable from random.

So we're going to have to use non-linearity, things that are called non-linear feedback shift registers, to create pseudo-random functions. There's many ways that we can create pseudo-random functions. We won't really get into that. But obviously that's what we want.

And then the last one is a bit tricky. And we will have an app that requires this way at the end. But this is infeasible given h of x to produce h of x prime, where x and x prime are-- and it gets a little bit fuzzy here-- are related in some fashion, right? So a concrete example of this is, let's say that x prime is x plus 1. So this is a reasonable example of this.

So what this says is you're just given h of x. It doesn't actually say anything about one-wayness yet. But you could assume, for example, that if this was a one-way hash function, that it would be possible to get x from h of x, correct? And let's keep that though. Hold that thought, all right? We're going to get back to it.

So if I'm just given the hash through some computation, it may be possible for me to create another hash, h of x prime, such that there's some relationship that I can prove or argue for between the strings that created the hashes, namely x and x prime, OK? That's what malleability is, right? Now you might just go off and say here's an x, here's a y, here's h of x, and here's h of y. These look completely random. And you might go off-- I'm being facetious here-- I say that y is x's third cousin's roommate's brother-in-law or something, right? I mean just make something up, right?

So clearly there's got to be a strong, precise relationship between x and y. If in fact you could do this and get y equals x plus 1, that'd be a problem, right? But if you are-- and then you can do this sort of consistently for different x's and y's, that would absolutely be a problem, right? But what you're really asking for-- and typically when you want non-malleability-- it's things where you have auctions, for example, where you are to be careful about making sure that you don't want to expose your bid. And so maybe what you're doing is exposing h of x.

You don't want somebody to look at your h of x and figure out how they could beat your bid by just a little bit. Or in case of Vickrey auctions, where the second highest bidder wins, now just be a little bit below you, right? So that's the kind of thing that you want to think about when it comes to non-malleability, or malleability, where you want a strong relationship between two strings that are related in some ordered fashion, like x equals-- x prime equals x plus 1, or just x prime equals 2 times x.

And you don't want to be able to-- you don't want the adversary to be able to discover these new strings. Because that would be the system, all right? So any questions about properties? Are we all good on these properties? All right, because I'm going to start asking you how to use them for particular applications, or what properties are required for certain applications, OK?

One last thing before we get there. I promised a slightly more detailed analysis of the relationships between these properties. So let's do that. Now if your just look at it, eyeball it, and you look at collision resistance and TCR, what can I say about the relationship between CR and TCR? If h is CR, it's going to be TCR, right? It's got to be TCR. It's a strictly stronger requirement.

But not reverse. And you can actually give a concrete example of a particular hash function that is TCR. I'm not going to go there. It's actually a little more involved than you might think it is, where a TCR hash function is not collision resistant. But you can see that examples such as these should exist, simply because I have a more stringent property corresponding to collision resistance as opposed to TCR, right?

So if you're interested in that particular example, you're not responsible for it, get in touch with me and I'll point you to a, like a three-page description of an example. So I didn't really want to go in there. But what I do want to do is talk about one-wayness and collision resistance. Because I think that's actually much more interesting, all right?

So if h is one-way-- any conjectures as to what the question mark is in the middle? Can I make strong statements about the collision resistance of a hash function, if I'm guaranteed that the hash function I have is a one-way hash function, or vice versa? Another way of putting it is, can you give me an example of, just to start with, a hash function which is one-way but not

TCR, not target collision resistant?

So I'm going to try and extract this out of you. This is somewhat subtle. But the way you want to think about this is, let's say that h of x is OW and TCR, OK? And so I have a bunch of inputs. And this is the output. And I get d-bits out. And I've got x1, x2, to xn, OK?

Now I've given this h-- I've been given this h which is one-way and TCR. It satisfies those properties that you have up there. In the case of one-way, I give you an arbitrary d-bit string. You can't go backwards and find a bunch of the xi's that produce exactly that d-bit string, all right? So it's going to be hard to get here.

But you're allowed now to give me an example. So this is some hash function that you can create, which may use h as well. And h is kind of nice because it has this one-way property. So let's say that we want to discover something where one-way does not imply TCR. So I want to cook up a hash function h prime such that h prime is one-way, but it's not TCR, OK?

The way you want to think about this is you want to add to h. And you want to add something to h such that it's still hard-- if you add h it's still hard to go from here to there. Because you've got to go deeper. If you add to, for example, the inputs of h. Or you could add to the outputs of h as well, or the outputs of the current h.

But you can basically go deeper, or need to go deeper in order to find the break one-wayness, in order to find an x, whatever you have, that produces the d-bit string that you have, right? So what's a simple way of creating an h prime such that it's going to be pretty easy to find targeted collisions even, not necessarily collisions, it's pretty easy to find targeted collisions, without breaking the one-way property of h? Yeah?

AUDIENCE: So if you have x sub i, if i odd then return h of x of i. So that's minus 1. So return the even group.

SRINIVAS DEVADAS: Sure. Yep.

AUDIENCE: Given x any x of i, you can usually find another x of i that was the same output? You can go backwards.

SRINIVAS DEVADAS: You can't go backwards. Yeah, that's good. That's good. I'm going to do something that's almost exactly what you said. But I'm going to draw it pictorially.

And what you can do, you can do a parity, like odd and even that was just described. And all I'll do is add a little [? XNOR ?] gate, which is a parity gate, to one of the inputs. So you have and b here. So I've taken x1, and I have a and b here. So I've added-- I can add as many inputs as I want to this function.

Oh I should mention by the way, h of x is working on arbitrary strings. And obviously I put in some number here that corresponds to n, which is a fixed number. So you might ask, what the heck happened here with respect to arbitrary strings? And there's two answers.

The first answer is, well, ignore arbitrary. And assume that you only have n-bit strings. And n this is really large number, right? And that may not be particularly satisfying.

The other answer is, which is more practical, which is what's used in practice, is that typically what happens is, you do have particular implementations of hash functions that obviously need to have fixed inputs, n, for example. And n is typically 512. It's usually the block size. And you chunk the input up into five 12-bit blocks.

And typically what you do is, you take the first five 12-bits, compute the hash for it. And then you can do it for the remaining blocks. And then you can hash all of them together, all right? So there's typically more invocations. I don't really want to get into it. But there's typically more invocations of h when the input would be 2 times n, or 3 times n, all right?

So we don't really need to go there for the purposes of this lecture. But keep that in mind. So we'll still stick with our arbitrary string requirement.

So having said that, take a look at this picture. And see what this picture implies. I have an h prime that I've constructed, right? Now if I look at h prime, and I give you an output for h prime-- so h prime now has, it's a function of a and b, and x2 all the way to xn, right? So it's got an extra input.

If I look at h prime, and I look at the output of h prime that is given to me, and I need to discover something that produces that, it is pretty clear that I need to figure out what these values are, all right? And I need to know what the parity of a and b is. And maybe I don't need to know exactly what a and b are, but I absolutely need to know what the parity of a and b are, because that's x1. And the one-way I'd break would require me to tell you what the value of x1 is, and the value of x2, and so on and so forth.

So it's pretty clear that h prime is one-way, right? Everybody buy that? h prime is one-way.

But you know what? I've got target collisions galore, right? All I have to do is flip-- I have a equals 1 and b equals 1. And I have a equals 0 and b equals 0. They're going to give me the same hash, right?

So trivial example, but that gets to the essence of the difference between collision resistance and one-wayness, target collision resistance and one-wayness, all right? So this is one-way but not TCR, simply because a equals 0, b equals 0 for arbitrary x's produce the same thing as a equals 1 and b equals 1, right? So those are collisions.

So admittedly contrived. But it's a counterexample. Counterexamples can be contrived. It's OK.

All right. So that was what happens with that. Let's look at one more interesting thing that corresponds to the other way, right? So what I want to show is that a TCR does not imply one-wayness. OK, so now I want an example where it is clear that I have target collision resistance, because I can just assume that. And we're going to use the same strategy.

I'm just going assume that I have an h that's target collision resistant. And I'm going to try and cook up an h prime that is not one-way. So I'm going to assume that in fact h is TCR and OW.

And I'm going to take away one of the properties. And if I take it one of the properties I have a counterexample, right? So think about how you could do this.

You have h as before. And I want to add some stuff around it such that it's going to be easy to discover-- for a large, for a constant fraction of hashes that I've given to me, not for any old hash. Because you can always claim that one-wayness is broken by saying I have x, I computed h of x, now I know what-- given h of x I know what x is. I mean you can't do that, right? So that's not breaking the one-wayness of it.

It's when you have an h of x and this is the first time you've seen it, you're trying to find what x is, right? So how would you-- how would you set it up so you break the one-wayness of h without necessarily breaking the target collision resistance of the overall hash function that you're creating? And you have to do something with the outputs, OK? You have to do something.

This is a little more involved. It's not as easy as this example. It's a little more involved. But any ideas? Yeah, go ahead.

AUDIENCE: So x is less than b returns x. If x is greater than b, return [INAUDIBLE].

SRINIVAS DEVADAS: Beautiful. Right. What color did you get last time?

AUDIENCE: Blue.

SRINIVAS DEVADAS: You got a blue last time? All right. Well you get a purple. You have a set. Actually we have these red ones that are precious, that are-- no, we don't. We chose not to do red. I don't know. There was some subliminal message I think with throwing red Frisbees that we didn't like.

But OK. So thank you. And h of x is simply something where I'm going to concatenate a zero to the x value and just put it out. And clearly this is breaking one-wayness because I'm just taking the input, I'm adding a zero to it, and shipping it out. So it's going to be easy to go backwards, right?

And this only happens if x is less than n, as the gentleman just said. Less than or equal to n in terms of the input length, OK? Otherwise I'm going to do h of x. So this is good news. Because I'm actually using the hash function in the case where I have a longer input string. This is bad news for one-wayness because I'm just piping out the input.

And so if I get an x, and I see what the x is out here, and let's just say for argument's sake that-- you could even say that n is going to be something that is less than d, which is the final output, which has d-bits. And so if you see something that h prime produces that's less than d-bits you instantly know that you can go backwards and discover what input produced that for the h prime, right? Because you just go off and you go backwards. This is what it tells you.

Now on the other hand if it's larger obviously you can't do that. But there's a whole lot of combinations that you can do that for. So this breaks one-wayness, OK?

Now you think about TCR. And what you want a show of course is that this maintains TCR. So that's the last thing that we have to show. We know that it breaks one-wayness. But if it broke TCR we don't quite have our example. So we want to show that it actually maintains TCR, which is kind of a weakish property that we need to maintain.

And the reason this maintains TCR is that there's really only two cases here obviously, corresponding to the if statement. And it's pretty clear that if x is less than or equal to n, clearly different x's produce different h prime x's, correct? Because I'm just passing along the x out to the output.

So if x is less than n I am going to get different hashes at the output. I'm just passing them out. So that's easy.

And for the other case, well I assume that h of x was CCR, correct? Because that was the original assumption, that I had h, which was CCR. So in both cases TCR is maintained because else h of x maintains TCR, all right? So again, a bit of a contrived example to show you the difference between these different properties so you know not to mix them up. You know what you want to ask for, what is required when you actually implement an application that depends on particular properties.

All right? Any questions so far about properties or any of these examples? We're going to dive in to using them.

OK. So start thinking computer security. Start thinking hackers, protecting yourself against the bad guys that are out there who are trying to discover your passwords, trying to corrupt your files, generally make your life miserable. And we'll start out with fairly simple examples, where the properties are somewhat obvious, and graduate to this auction bidding example which should be sort of the culmination of at least this part of the lecture.

And depending on how much time I have I'll tell you a little bit about how to implement hash functions. But I think these things are more important from a standpoint of giving you a sense of cryptographic hashes. All right. Password storage.

How many of you write your password in an unencrypted text file and store it in a readable location? There you go, man. Thank you for being honest. And I do worse. Not only do I do that, I use my first daughter's name for four passwords. I won't tell you what the name is.

So that's something that we'd like to fix, right? So what do real systems do? Real systems cannot protect against me using my first daughter's name as a password, right? So there's no way you can protect against that.

But if I had a reasonable password, which had reasonable entropy in it-- so let's assume here that we have reasonable entropy in the password. And you can just say 128-bits. And it's not a lot, right? 128-bits is 16 characters, OK? And you don't have to answer this-- how many of you have 16 characters in your password?

Oh I'm impressed. OK. So you've got 128-bits of entropy. But the rest of you, forget it. This is not going to help you, OK? But what I want, assuming you have significant entropy in your password-- because otherwise, if there's not enough entropy you can just enumerate all possible passwords of eight letters. And it's not that much. It's 2 raised to 50, what have you. And you can just go off. And none of these properties matter.

You just-- you have your h of x. It's public. We'll talk about how we use that in a second. But clearly if the domain is small you can just enumerate the domain. So keep that in mind.

I talked about h of x, and it's obviously going to be relevant here. But suppose I wanted to build a system, and this is how systems are built, ETC slash password file, assuming you have long passwords it does it this way, otherwise it needs something that's called a salt. But that's 6, 8, 57 and we won't go there. So we just assume a large entropy.

What is it that a system can do? What can it store in order to let you in, and only let you in when you type your password, and not let some bogus password into the system? Or somebody with a bogus password into the system. Yeah, go ahead.

AUDIENCE: If you capture the password when you enter it and compare it to what's stored--

SRINIVAS DEVADAS: Yes.

AUDIENCE: If it's a one-way hash you know you have what the correct password is.

SRINIVAS DEVADAS: That's exactly right. That's exactly right. So it's a really simple idea, a very powerful idea. It, as I said, assumed that the entropy-- and I'm belaboring the obvious now-- but it is important when you talk about security to state your assumptions. But you do not store password on your computer. And you store the hash of the password.

Now why do I store my password on the computer? Because this is so inconvenient, right? So this is what the system does for me. But the fact of the matter is, if I lose my password, this doesn't help me. Because what the system wants you to do is choose a password that is long enough, and the h is one-way. So anybody who discovers h of PW that is publicly readable cannot discover PW, all right? That's what's cool about this.

How do you let the person log in? Use h of PW to compare against h of PW prime, which is what is entered, where PW prime is the typed password.

And clearly what we need is the disclosure of h of PW should not reveal PW. So we definitely need one-wayness. What about-- what about collision resistance? Our target collision resistance?

Think practitioner now, right? Are we interested in this hash function being collision resistant? What does that mean in this case? Give me the context in this particular application? Yeah, go ahead.

AUDIENCE: It means that someone entering a different password will have the same hash [INAUDIBLE].

SRINIVAS DEVADAS: Exactly. So it means that what you have is a situation where you do not reveal-- and so what might happen is that h of PW prime equals h of PW. But h of PW equals h of PW prime. But PW is not equal to PW prime. What you have is a false positive.

Someone who didn't know your password but guessed right-- and this is a 128-bit value, and they guessed right-- is going to get it. You don't particularly care of the probability of this occurrence. It's really small. Typically you're going to have systems that lock you out if you try 10 tries that occurs one, two, wrong passwords, right?

So really in systems you do not require-- you do want to build systems that have minimal properties with respect to the perimeters that are used. So from a system building standpoint just require OW. Don't go overboard. Don't require collision resistance or TCR, OK?

Let's do a slightly different example. Also a bit of a warm-up for what's coming next, which is a file modification detector. So for each file F, I'm going to store h of F. And as securely. So you assume that this means that h of F cannot be modified by anybody, h of F itself.

And now we want to check if F is modified by re-computing h of F. Which could be, this could be modified. So this could actually be F prime. You don't know that.

You have a file. It's a gigabyte. And somebody might have tampered with one of the bits in the file. All you have is a d-bit digest that corresponds to h of F that you stored in a secure location. And you want to check to see, by re-computing h of F, the file that is given to you, and comparing it with what you've stored, the h of F that you've stored.

And so what property do we need in order to pull this off? Of hash functions. What precisely do we need to pull this off? What is the adversary trying to do? And what is a successful break?

A successful break is if an adversary can modify the file and keep h of F the same, right? That would be a successful break, right? Yup. Go ahead.

AUDIENCE: TCR.

SRINIVAS DEVADAS: TCR? Yeah, absolutely. You need TCR. So you want to modify the file. So you're given that the file-- the adversary is given the file, which is the input to the hash, and is going to try and modify-- modify the file, right?

So let's do a couple more. And we're going to advance our requirements here a little bit. So those two are basic properties. I want to leave this up there.

We're going to do something that corresponds to digital signatures. So digital signatures are this wonderful invention that came out of MIT in a computer science laboratory-- again, Ron Rivest and collaborators-- which are a way of digitally signing a document using a secret key, a private key. But anybody who has access to a public key, so it could be pretty much anybody, could verify the authenticity of that signature, right? So that's what a digital signature is.

So we're going to talk about public cryptography on Thursday, in terms of how you could build systems or encryption algorithms that are public key algorithms. But here I'll just tell you what we want out of them. Essentially what we have here in the case of signatures, we actually want to talk about encryption here, are-- there's two keys associated with a public key system.

Anybody and everybody in the system would have a public key that you can put on your website. And you also have a secret key-- that's like your password-- that you don't want to write down, you don't want to give away, because that's effectively your identity. And what digital signatures respond to are that you have two operations. You have signing and verification.

So signing means that you create a signature sigma that is the sign using your private key, your secret key, off a message M. So you're saying this is this message, it came from me, right? That's what signing means. You have this long message and you sign it at the bottom. You're taking responsibility for the contents of that message.

And then verification is you have M sigma and a public key. And this is simply going to output true or false. And so the public key should not reveal any information about the secret key. And that's the challenge of building PKI systems, that we'll talk about in some detail next time. But we don't need to think about that other than acknowledging it today.

So the public and private key are two distinct things, neither one of which reveals anything about the other. Think of them as completely distinct passwords. But they happen to be mathematically related. That's why this whole thing works. And that mathematical relationship we'll look at in some detail on Thursday.

But having said that, take a look at what this app is doing for us, right? This is a security application. And I haven't quite gotten to hash functions yet. But I'll get to it in just a minute.

But what I want to do is emphasize that there's two operations going on. One of which is a signature, which is a private signature, in the sense that it's private to me, if I'm Alice. Or private to Alice. And you're using secret information on this public message, M, because that's going to be publicized. And you're going to sign the public message.

And then anybody in the world who has access to Alice's public key is going to be able to say, oh I'm looking at the signature, which is a bunch of bits. I'm looking at the message, which is a whole lot of bits. And I have this public key, which is a bunch of bits. And I'm going to be able to tell for sure that either Alice signed this message, or Alice did not sign this message.

And the assumption here is that Alice kept her private key secret. And of course, what I just wrote there, that the public key does not reveal anything about the secret key, OK? So that's digital signatures for you, in a nutshell. And when you do MIT certificates you're using digital signatures a la Rivest-Shamir-Adleman, the RSA algorithm. So you're using this all the time, when you click on 6.046 links, for example.

And what happens is M is typically really large. I mean it could be a file, right? It could be a large file. And you don't necessarily want to compute these operations on large files. So for convenience, what happens is you end up hashing the file. And for large M it's easier to sign h of M. And so replace the M's that you see here with h of M, all right?

So now that we're given that we're going to be doing h of M in here, think about what we wanted to accomplish with M, right? I told you what we wanted to accomplish with M. There's a particular message. I'm Alice. I'm going to keep my secret key secret. But I want to commit to signing this message M, all right? And I want to make sure that nobody can pretend to be me who doesn't know my secret key. And nobody does.

So if I'm going to be signing the hash of the message, now it comes down to today's lecture. I'm signing the hash of the message h of M. What property do I require of h in order for this whole thing to work out? Yeah, go ahead.

AUDIENCE: Is it non-malleability?

SRINIVAS DEVADAS: Non-malleability, but even before that-- suppose-- absolutely, but non-malleability is kind of beyond one of these properties over on the right. You're on the right track, right? So do you want to give me a different answer? You can give me a different answer.

AUDIENCE: Oh, I'm not sure.

SRINIVAS DEVADAS: OK. What? Yeah, back there.

AUDIENCE: I think you wanted to one-way because otherwise you could take that signature and find another message that you could credit.

SRINIVAS DEVADAS: I can make M public. I can make M-- M can be public. And h of M is public. So one-wayness is not interesting for this example if M is public. And we can assume that M eventually gets public. Because that's the message I'm signing, right? I can also put M out.

So I want the relationship-- I want you to focus on the relationship between h of M and M and tell me what would break this system. And you're on the right track. Yeah, go ahead. Or way back there. Yeah, sorry about that.

AUDIENCE: TCR.

SRINIVAS DEVADAS: TCR. Why TCR?

AUDIENCE: [INAUDIBLE].

SRINIVAS DEVADAS: So I have M. So what happens here-- I should write this out. I'm given-- as an adversary I have M and h of M. It is bad if Alice signs h of M, but Bob claims Alice signed M prime. Because h of M equals h of M prime, right? That is bad.

So the M is public-- could you stand up? M is given. There's a specific M, and a specific h of M in particular, that has been exposed. And h of M is what was used for the signature. So you want to keep h of M the same. It's a specific one. So it's not collision resistance, it's target collision resistance, because that's given to you.

And you want to keep that the same. But you want to claim that oh, you promised me $10,000, not $20, right? If you can do that, you signed saying you want to pay $10,000, not $20, then you've got a problem.

So your thing is very close. It's just that it doesn't need to be a strong relationship between the 10,000 or the 20. I mean I give you a concrete example of that. But it could be more, it could be less. Anything that is different from what you signed, be it with the numerical relationship or not, would cause a problem and break this scheme, all right? Are we good?

All right, one last example, the most interesting one. And as I guessed I'm probably not going to get to saying very much about how cache functions are implemented. But maybe I'll spend a minute or two on it. So let's do this example that has to do with commitments.

Commitment is important, right? You want to commit to doing things. You want to keep your promises. And in this case we have a legal requirement that you want to be able to make people honor their commitments, and not weasel their way out of commitments, right? And we want to deal with this computationally.

And let's think about auctions. So Alice has value x, e.g. an auction bid. Alice computes what we're going to call C of x, which is a commitment of x, and cements it, right? C of x, C of x is-- let's assume that the auctioneer, and perhaps other auctionees as well, see C of x. You have to submit it to somebody, right? So you can assume that that's exposed.

And what is going to happen is, when bidding is over Alice is going to open-- so this is-- C of x can be thought of as sealing the bid. So that's the commitment. You're sealing the-- you're making a bid and you're sealing it in an envelope. You've committed to that. That's obviously, what happens in real life without cryptography, but we want to do this with cryptography, with hash functions.

And so now Alice opens C of x to reveal x. So she has to prove that in fact x was her bid. And that it matches what she sealed. When you open it up, think about it conceptually from a standpoint of what happens with paper, and then we have to think about this computationally and what this implies, right?

So again I'll do a little bit of set up. And then we have start talking about the properties that we want for this particular application. So there are a bunch of people who are doing bidding for this auction. I don't-- I want to be the first-- I don't want to spend a lot of money. But I want to win. All of us are like that, right?

If I know information about your bid, that is obviously a tremendous advantage. So clearly that can't happen, right? If I know one other person's bid I just do plus 1 on that. If I know everybody else's I just do plus 1 on the maximum. So clearly there's some secrecy that's required here, correct?

So C of x is going to have to do two things. It can't reveal x. Because then even maybe the auctioneer is bad. Or other people are looking at this. And you can just assume that C of x is-- the C of x's are all public.

But I also need a constraint that's associated with C of x that corresponds to making sure Alice is honest, correct? So I need to make Alice commit to something, right? So what are the different properties of the hash function that if I use h of x here, that I'd want h to satisfy in order for this whole process to work like it's supposed to work with paper and envelopes? Yeah, go ahead.

AUDIENCE: It has to be one-way [INAUDIBLE].

SRINIVAS DEVADAS: It has to be one-way. And explain to me-- so I want a description of it has to be one-way, because why?

AUDIENCE: Because you want all the c x's to be hidden from all the other options.

SRINIVAS DEVADAS: Right. C of x should not reveal x, all right? All right. That's good. Do you have more? It has to be collision resistant. OK. I guess.

A little bit more. You're getting there. What-- why is it collision resistant?

AUDIENCE: Because you want to make sure that Alice, when she makes a bid that she commits that bid. If she's not going to resist it then she could bid $100 and then find something else.

SRINIVAS DEVADAS: That's exactly right. So CR, because Alice should not be able to open this in multiple ways, right? And in this case it's not TCR in the sense that Alice controls what her bids are. And so she might find a pair of bids that collide, correct? She might realize that in this particular hash function, you know $10,000 and a billion dollars collide, right?

And so she figures depending on what happens, she's a billionaire, let's assume. She's going to open the right thing. She's a billionaire, but she doesn't necessarily want to spend the billion, OK? So that's that, right? But I want more. Go ahead.

AUDIENCE: You don't want it to be malleable. Assuming that the auctioneer is not honest because you don't want to accept a bribe from someone and then change everyone else's bid to square root of whatever they bid.

SRINIVAS DEVADAS: That's exactly right. Or plus 1, which is a great example, right? So there you go. I ran out of Frisbees. You can get one next time.

So yeah, I don't need this anymore. You're exactly right. There's another-- it turns out it's even more subtle than what you just described. And I think I might be able to point that out to you. But let me just first describe this answer, which gives us non-malleability.

So the claim is that you also want non-malleability in your hash function. And the simple reason is, given C of x-- and let's assume that this is public. It's certainly public to the auctioneer, and it could be public to the other bidders as well.

Because the notion of sealing is that you've sealed it using C of x. But people can see the outside of the envelope, which is C of x. So everyone can see C of x. You still want this to work, even though all other bidders can see C of x.

So given C of x, should not be possible to produce C of x plus 1. You don't know x is. But if you can produce C of x plus 1, you win, all right? And so that's the problem.

Now it turns out you now say OK, am I done? I want these three properties. And I'm done, right? There's a little subtlety here which these properties don't capture. So that's why there's more here. And I don't mean to titillate, because I'll tell you what is missing here.

But let's say that I have a hash function that looks like this. And this here is non-malleable. It is collision resistant. And it's one-way, all right? So h of x has all these wonderful properties, all right?

I'm creating an h prime x that looks like this, which is a concatenation of h of x, and giving away the most significant bit of x, which is my bid, right? I'm just giving that away, right? The problem here is that we haven't really made our properties broad enough to solve this particular application to the extent that there's contrived cases where these properties aren't enough, OK?

And the reason is simple. h prime x is arguably NM, CR, and OW. And I won't go into to each of those arguments. But you can think about it, right?

If I'm just giving you one bit, there's 159 others, there's a couple of hundred others, whatever it is that I have in the domain. It's not going to be invertible. h prime x is not going to be invertible if h of x is not invertible. h prime x is not going to be breakable in terms of collision resistance if h of x is not breakable, and so on and so forth.

But if I had a hash function like that, is it a good hash function for my commitment application? No, obviously not. Because if I publicize this hash function-- remember, everything is public here with respect to h and h prime-- you are giving away the most significant that corresponds to your bid in this particular hash function, right?

So you really need a little bit more than these for secrecy, for true secrecy. But in the context of this example, I mean it's common sense that you would not use the hash function like that, right? So it's not that there's anything profound here. It's just that I want to make sure that you understand the nuances of the properties that we're requiring.

We had all the requirements corresponding to the definitions of NM and CR and OW. And you need a little bit more for this example, where you have to say something, perhaps informally, like the bits of your auction are scrambled in the final hash output, which most hash functions should do anyway, and h of x will definitely do. But you kind of unscrambled it by adding this little thing in here, corresponding to the most significant thing, all right?

So I'll stop with that. Let me just say that the operation-- or sorry, the work involved in creating hash functions that are poly-time computable is research work. People put up hash functions and they get broken, like MD4 was put up in '92 and then got broken, SHA-1 and so on and so forth. And so I just encourage you to look up SHA-3 and just take a quick scan and what the complexity of SHA-3 is with respect to computing the hash given an arbitrary string, all right?

I'll stick around for questions.