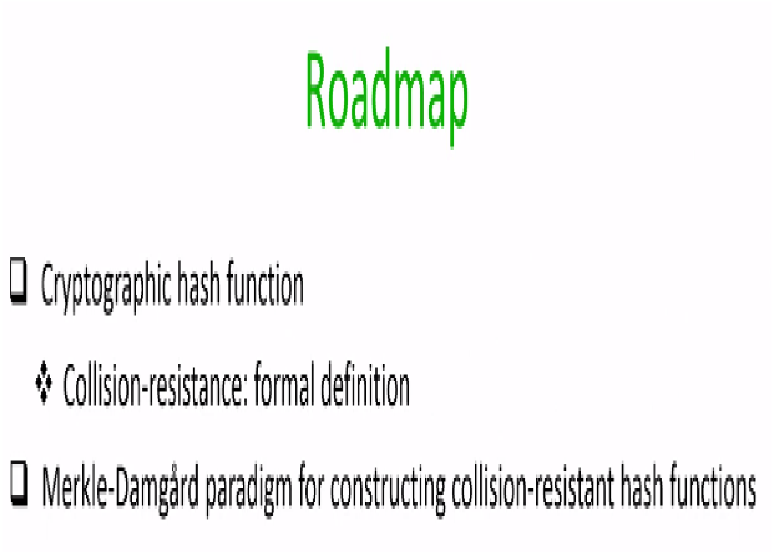
Foundations of Cryptography

Prof. Dr. Ashish Choudhury
(Former) Infosys Foundation Career Development Chair Professor
International Institute of Information Technology – Bangalore

Lecture – 27
Cryptographic Hash Functions - Part I

Hello everyone, welcome to this lecture. Just to recall in the last lecture, we had seen how to construct message authentication codes which are secure even against a computationally unbounded adversary.

(Refer Slide Time: 00:46)

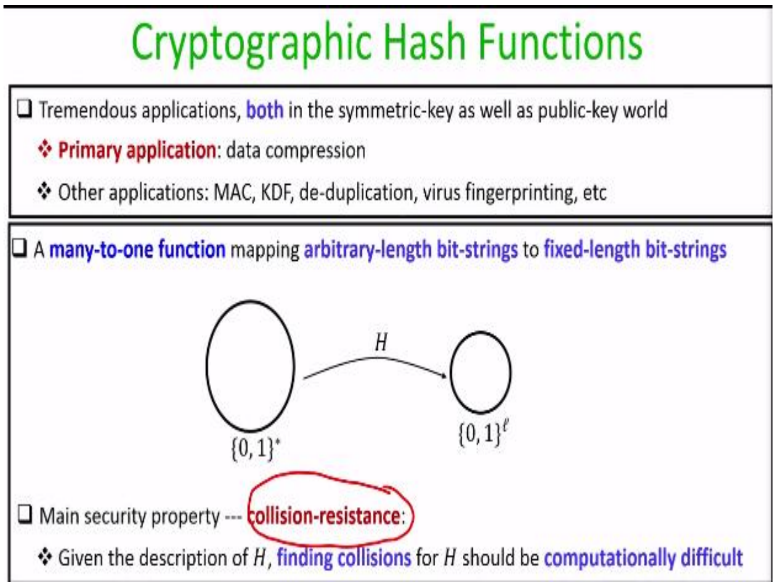


Roadmap

- Cryptographic hash function
  - Collision-resistance: formal definition
- Merkle-Damgård paradigm for constructing collision-resistant hash functions

So, in this lecture, we will introduce another interesting cryptography primitive called cryptographic hash functions, which has got several applications with one of the applications being the construction of efficient message authentication codes. So, we will introduce the formal definition of cryptographic hash function and we will also formally define what exactly we mean by the collision-resistance property of cryptographic hash functions and then we will discuss the Merkle-Damgard paradigm, which is an interesting paradigm and used for construction of several practical collision-resistant hash functions.

(Refer Slide Time: 01:18)



Cryptographic Hash Functions

- Tremendous applications, both in the symmetric-key as well as public-key world
  - Primary application: data compression
  - Other applications: MAC, KDF, de-duplication, virus fingerprinting, etc
- A many-to-one function mapping arbitrary-length bit-strings to fixed-length bit-strings

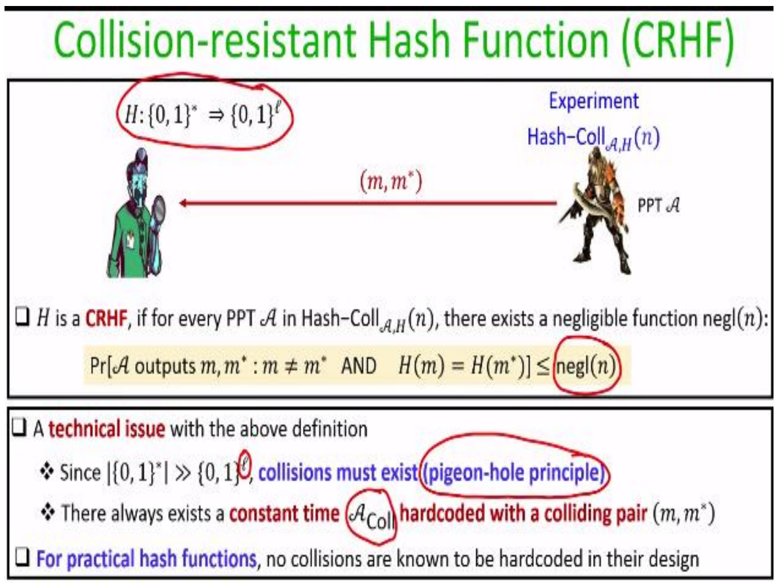
$$\begin{tikzcd}\{0,1\}^* \arrow[r, "H"] & \{0,1\}^\ell\end{tikzcd}$$

- Main security property --- collision-resistance:
  - Given the description of H, finding collisions for H should be computationally difficult

So let us begin with the definition of cryptographic hash functions. So, this primitive has got tremendous applications both in the symmetric key as well as in the public key setting and the primary application of this cryptographic primitive is data compression and it has got several other applications like construction of message authentication code, as a key derivation function, for de-duplication purpose, virus fingerprinting, and so on. So on a very high level, a cryptographic hash function is a many-to-one function, mapping arbitrary-length bit strings to fixed-length bit strings.

So the domain is the set of bit strings which can be of any length and output is always of a fixed length say *l* where *l* is the function of the security parameter and property wise we require many properties from this cryptographic hash function, but main property which we are interested in and which we will discuss in this course is the collision-resistance property and on a very high level or informally what exactly collision resistance means that an adversary or an algorithm, which is computationally bounded even though if it is knowing the description of this function H should not be able to find out a collision or a pair of distinct inputs, which gives you the same hash value, right except with some negligible probability. That means, it should be very difficult computationally to find collisions in a reasonable amount of time.

(Refer Slide Time: 02:49)



Collision-resistant Hash Function (CRHF)

$H: \{0,1\}^* \Rightarrow \{0,1\}^\ell$

Experiment

Hash–Coll$_{\mathcal{A},H}(n)$

(m, m^*)

PPT $\mathcal{A}$

- H is a **CRHF**, if for every PPT $\mathcal{A}$ in Hash–Coll$_{\mathcal{A},H}(n)$, there exists a negligible function negl(n):

  Pr[$\mathcal{A}$ outputs $m, m^* : m \neq m^*$ AND $H(m) = H(m^*)$] $\leq$ negl(n)

- A **technical issue** with the above definition
  - Since $|\{0,1\}^*| \gg \{0,1\}^\ell$, **collisions must exist** **(pigeon-hole principle)**
  - There always exists a **constant time** $\mathcal{A}_{\text{Col}}$ **hardcoded with a colliding pair** (m, m^*)
- **For practical hash functions**, no collisions are known to be hardcoded in their design

So, let us formally define what exactly we mean by collision resistance. So, that is modelled by an experiment and the experiment we have given the description of a publically known hash function, right? And name of the experiment is hash-collision experiment and we have a polynomial time adversary. Basically, the goal of the adversary is to come up with a pair of messages m, m* from the domain of this hash function.

The security definition here is that we say that the function H is a collision resistant-hash function or CRHF if for every polynomial time adversary participating in this experiment, the probability that the adversary could come up with a colliding pair, namely with a pair of distinct inputs m, m* such that both m as well as m* gives you the same hash value is upper bounded by some negligible function, right. So basically the goal of the challenger, the goal of the adversary here is to take the description of your hash function and come up with a pair of colliding inputs.

If it is able to do that with a non-negligible advantage, then we say that our function H is not collision-resistant hash function, otherwise we say that the function H is a collision-resistant hash function. Notice that the function H is not a keyed function, it is an unkeyed function and the function H is a deterministic function. There is no internal randomness present inside the function H, right. So even though the function H is deterministic, there is no key. The challenge for the adversary is to come up with a pair of colliding inputs.

It turns out that there is a slight technical issue with above definition in the sense that if the above is the definition of collision-resistant hash function, then we cannot define any function

H or we cannot construct any function H which satisfies the above definition. This is because as I said that the domain of the function H is a set of all bit strings and which is significantly large than the size of the co-domain because your co-domain consist of only strings of length l bits and that is why the function H is a many-to-one function.

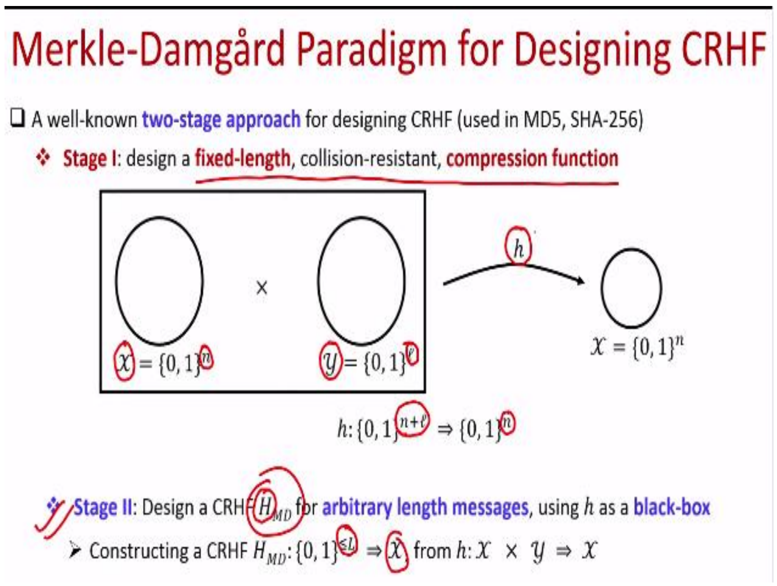
As a result, there are always collisions which are present in the function H, which follows from your pigeon-hole principle because if you have a many-to-one function, then definitely you will have multiple inputs x, x* or m, m* which have the same hash output. There always exist adversary which I say a collision adversary, which could be hardcoded with such a colliding pair of inputs m, m*.

That means, if such an adversary A_{coll} participates in this hash-collision experiment, then it can simply output the message pair m, m*, which is hardcoded in the adversary. The adversary do not have to do any step, it is a constant time attack. That means, fundamentally the way we have defined a collision-resistant property, it is not possible to satisfy the definition against any construction of hash function, right. However, interestingly, it turns out that most of the practical instantiations of hash function some of which we are going to discuss, there are no collisions which are hardcoded in their design.

That means, we believe that there exists no adversary which already knows or which is hardcoded with a pair of inputs m, m* which constitutes a collision for the corresponding hash function. So, that is why technically even though there is a challenge associated with the definition, we stick to this definition of collision-resistant hash function. I also stress that the security property of the collision-resistant property does not demand that there should not be any collision in the function H because by design itself, it is a many-to-one function.

As I said, by pigeon-hole principle it follows that there will be several collisions, which are present with respect to the function H. The challenge is to design a poly-time algorithm or a computationally efficient algorithm which when given the description of the function H could come up with at least one such pair of collision with a non-negligible advantage that is a security requirement or that is what we mean by the collision-resistant property.

(Refer Slide Time: 07:06)



Merkle-Damgård Paradigm for Designing CRHF

- ❑ A well-known **two-stage approach** for designing CRHF (used in MD5, SHA-256)
  - ❖ **Stage I**: design a **fixed-length**, collision-resistant, **compression function**

$\mathcal{X} = \{0,1\}^n$
×
$\mathcal{Y} = \{0,1\}^\ell$
h
$\mathcal{X} = \{0,1\}^n$

$h: \{0,1\}^{n+\ell} \Rightarrow \{0,1\}^n$

- ❖ **Stage II**: Design a CRHF H_{MD} for **arbitrary length messages**, using h as a **black-box**
  - ➢ Constructing a CRHF $H_{MD}: \{0,1\}^{\le L} \Rightarrow \mathcal{X}$ from $h: \mathcal{X} \times \mathcal{Y} \Rightarrow \mathcal{X}$

So now, we have the definition of collision-resistant hash function. So we will be interested to see whether indeed it is possible to design such functions or not. What we are going to discuss is a well known paradigm which is called as Merkle-Damgard paradigm and it is a very well known 2-stage approach which is used for the design of several practical instantiations of collision-resistant hash functions such as the MD5 hash function and several hash functions in the SHA-256 family. So as I said, it is a 2-stage approach for constructing a collision-resistant hash function.

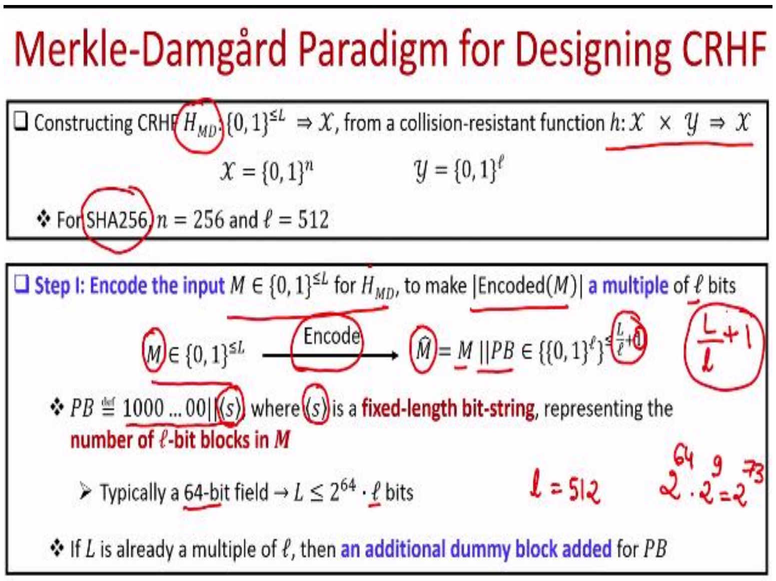
So in stage 1, what we do is we aim to design a fixed-length collision-resistant compression function and why it is fixed length because unlike a collision-resistant hash function, there the domain could consist of any string of any length, here the domain is fixed in the sense that it can take inputs only of size n+ *l* bits. It is called a compression function because it takes an input of size n+ *l* bits and produces an output of size of n bits. So definitely the output size is shorter or less than the input size and that is why it is a compression function.

So pictorially, you can interpret that we are interested in stage 1 to construct some function h, which takes an input of size n+ *l* bits which can be passed into 2 input halves, the first half of size n bits and the second half of size *l* bits. So that is why you can interpret the domain of this function h to be the Cartesian product of an x set which consists of strings of length n bits and another set y which consists of strings of length *l* bits and given a string of length n+ *l* bits as the input, the goal of this computation function should be to produce and output of size n bits such that this h function should be a collision-resistant function. That means, given the description of this function h, it should be difficult to come up with a pair of collision in polynomial amount of time with a significant probability. Once we have such a fixed-length compression function in stage 2, what we do is we apply this well-known Merkle-Damgard paradigm to construct a collision-resistant hash function which we denote by H_{MD}, which can take any string as input of length up to say L bits.

There is no restriction on the input size. The input could be of size 1 bit, 2 bit or it could be any string of length up to L bits, and it gives you an output belonging to the set x, right. So, that is what we will do in stage 2. The construction in stage 2 is a very generic construction, in the sense it can take any fixed-length compression function without going into the underlying details of that compression function and magically it will give you the collision-resistant function H_{MD} which you are interested to construct.

So, what we are going to discuss now is what we exactly do in stage 2. That means, we will assume we are given a fixed-length collision-resistant compression function h and then we will see how we apply the Merkle-Damgard paradigm and get the collision-resistant hash function H. Later in the next lecture, we will see what exactly we do for stage 1 that means how exactly we construct this candidate h compression function.

(Refer Slide Time: 10:31)



Merkle-Damgård Paradigm for Designing CRHF

- Constructing CRHF $H_{MD}: \{0,1\}^{\leq L} \Rightarrow \mathcal{X}$, from a collision-resistant function $h: \mathcal{X} \times \mathcal{Y} \Rightarrow \mathcal{X}$

  $\mathcal{X} = \{0,1\}^n$ $\mathcal{Y} = \{0,1\}^\ell$

  - For SHA256 $n = 256$ and $\ell = 512$

- **Step I: Encode the input** $M \in \{0,1\}^{\leq L}$ for H_{MD}, to make $|\text{Encoded}(M)|$ **a multiple** of ℓ bits

  $$M \in \{0,1\}^{\leq L} \xrightarrow{\text{Encode}} \hat{M} = M \,||\, PB \in \{\{0,1\}^\ell\}^{\leq \frac{L}{\ell}+1}$$

  $\frac{L}{\ell}+1$

  - $PB \overset{\text{def}}{=} 1000\ldots00 \,||\, \langle s \rangle$ where $\langle s \rangle$ is a **fixed-length bit-string**, representing the **number of ℓ-bit blocks in M**
    - Typically a 64-bit field $\rightarrow L \leq 2^{64} \cdot \ell$ bits

      $\ell = 512$ $2^{64} \cdot 2^{9} = 2^{73}$
  - If L is already a multiple of ℓ, then **an additional dummy block added** for PB

So our goal is the following. We are given a fixed-length compression function, taking inputs from the Cartesian product of x set and y set and giving you an output on the x set and the x set basically consist of strings of length n bits and the y set consists of strings of length *l* bits and our goal is to construct this function H, which can take any binary string of length up to L bits and give you an fixed size output, namely a string of length *l* bits. So if you are wondering what exactly are the values of n and *l*.

For the practical instantiations of hash function, so for your information for the SHA256 hash function, the value of n is 256 and the value of *l* is 512 bits, right. So the first thing that we do while applying the Merkle-Damgard paradigm is that we take the input M for the hash function H which we are interested to construct, and this input M is a binary string of length up to length L baits. So, we apply some encoding function here and the encoding is done to ensure that the encoded input is a multiple of *l* bits.

The reason we want to ensure that the encoded input is a multiple of *l* bits is that when we are going to apply the Merkle-Damgard paradigm, we are going to divide our encoded input into several blocks of *l* bits and we will be iteratively applying the fixed-length compression function. So, pictorially you can imagine that you are given this input, we apply some publicly known deterministic encoding function.

Encoded output is denoted by this $\widehat{M}$ which consists of the original M and concatenated with some padded bits and together this original M concatenated with the padded bits now will consist of several blocks of *l* bits and the number of such blocks of *l* bits will be (L/*l*) + 1. So, you might be wondering that why this +1, so this will be clear soon. So, what exactly is the form of this padded bits? Well, the padded bits is defined as follows. It will start with 1 followed by the required number of 0 concatenated with the binary representation of the number of *l* bit blocks which are present in the original M, right.

So, you have the original M which is the actual input which you want to hash using the H function which you are interested to construct. So, we count the number of *l* bit blocks which are present in the non-encoded input and the binary representation of the number of such blocks is this representation, namely the number <s> (within the angle brackets). So, the padded bit which we are actually appending to the bits of the message which we want to hash is of this form, we have 1 followed by the required number of 0 followed by the binary representation of the number of blocks of *l* bits which are present in M and typically the number of bits which are allocated for this binary representation of the number of *l* bit blocks present in M is a 64 bit field, but you can have this field consisting of more number of bits, but

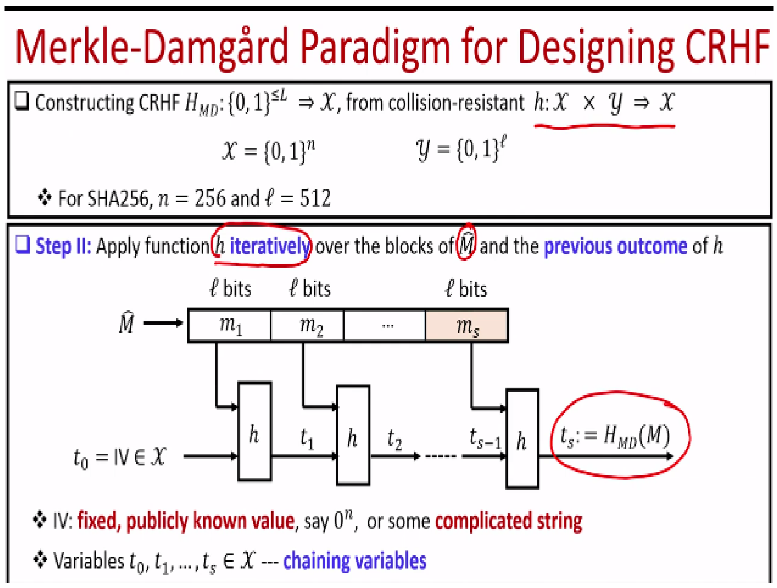
I am quoting this number with respect to one of the practical instantiations of hash functions, namely the SHA function.

So, that means, you could have up to 2^{64}. l number of blocks present in your original input M and each block consists of l bits. So, that gives you an upper bound on the maximum size of the message, which you can hash using this hash function H which you are interested to construct, right. So, again, if I take the example of SHA256, my l is 512 and I could have up to 2^{64} such blocks. So that gives you the maximum length of the message which you can encode, right, 2^{73} - that much length string you can hash using the function H_{MD} which you are interested to construct.

Interestingly, if your message M which you want to hash is consisting of number of blocks so that its length is already a multiple of l bit that means you do not need to actually do any padding, but then how exactly the receiver who is going to receive the message will come to know whether the padding has happened or not. So in case if the message length is already a multiple of l, then what exactly we do is we do the padding, where padding basically consist of a full dummy block starting with the representation 1000s and binary representation of the number of blocks of l bits which are already present in the message.

So that is why it respective of whether the message length is already a multiple of l bits or not, we actually do the padding and that is why this +1 is present in the number of blocks of l bits in the encoded input, right. So if the M is already a multiple of l, then you have these many (L/l) numbers of blocks of l bits and we are actually doing a padding, namely we are adding a full dummy block. So that is why the number of blocks of l bits which could be present in the encoded input is actually this ($L/l + 1$), right.

(Refer Slide Time: 16:16)



Merkle-Damgård Paradigm for Designing CRHF

- Constructing CRHF $H_{MD}: \{0,1\}^{\leq L} \Rightarrow \mathcal{X}$, from collision-resistant $\underline{h: \mathcal{X} \times \mathcal{Y} \Rightarrow \mathcal{X}}$

$$\mathcal{X} = \{0,1\}^n \qquad \mathcal{Y} = \{0,1\}^\ell$$

  - For SHA256, $n = 256$ and $\ell = 512$

- Step II: Apply function h iteratively over the blocks of $\hat{M}$ and the previous outcome of h

  - IV: fixed, publicly known value, say 0^n, or some complicated string
  - Variables $t_0, t_1, \ldots, t_s \in \mathcal{X}$ --- chaining variables

So now once you have the encoded input which you want to hash, right, so the way we compute the hash of the encoded input is that we iteratively apply the fixed-length collision-resistant compression function that we are available with and we do it iteratively in the sense we apply invocation of h over the current block of the encoded input and the previous outcome of the instantiation of the fixed-length collision function. So imagine that this is your pictorial representation of the encoded input consisting of several blocks of *l* bits.

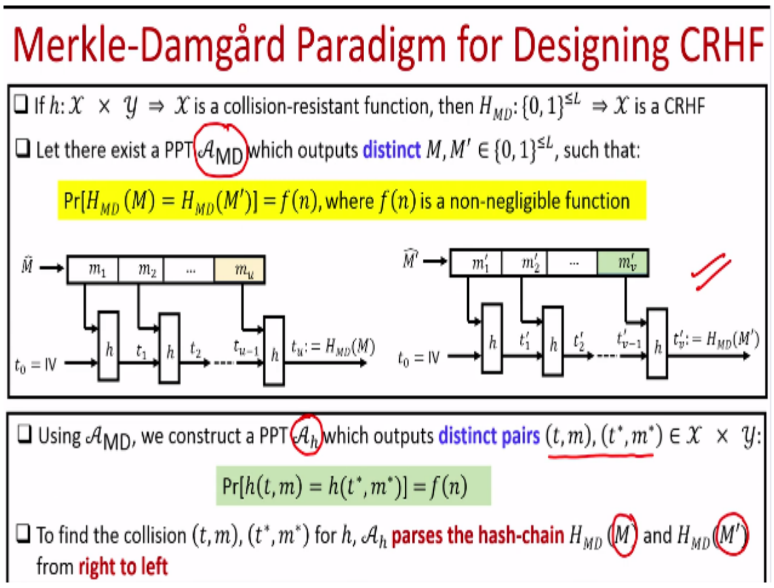
I am highlighting the last block because the last block may not be fully consisting of the input bits, it could be consisting of the padded bits as well, right. So, other than that, all the remaining blocks are not highlighted. So the way we iteratively apply the h function here is as follows. So, the first invocation is on the first block of *l* bits of the encoded input and along with an IV, which we denote as t_0 belonging to the set x, we will soon see what exactly is the value of IV and now you see the invocation of h, it takes an input of size *l*+n bits, right.

It is basically taking an input from the x set and it is taking an input from your y set, which satisfies the semantic of my h function and it will give you an output belonging to the set x. So the output that comes out of the first invocation of the h function is denoted by t_1, it will be of size n bits and we take the next chunk of *l* bits from the encoded input and apply the next invocation of the h function and output is denoted as t_2, which will be again of size n bits and we continue like that till we are done with the last invocation of the h function, which operates over the last block of the encoded input of size *l* bits.

The outcome of the previous invocation of the h function and the final output of the overall hash function H which we have constructed for the input M is considered to be the outcome of the final invocation of the h function. So that is how we are going to hash the message M, right. So that is why the function h here is applied iteratively and this also tells you why exactly we want to encoded input to be a multiple of *l* because in each iteration, we take one chunk of *l* bits and apply the h function on the previous outcome of the h function, right.

So now you might be wondering what exactly is an IV? Is it some random entity or not? So as I said the hash functions are deterministic functions, so IV is not a random value, it is a fixed publicly known value, you can take it say all zeros which is set once for all or for some practical instantiations of the hash function, this IV set is a very complicated string and this intermediate variables t_0, t_1, t_2, t_{s-1}, t_s, which we actually obtain along this sequence of chaining process, they are called as chaining variables.

(Refer Slide Time: 19:40)



Merkle-Damgård Paradigm for Designing CRHF

- If $h: \mathcal{X} \times \mathcal{Y} \Rightarrow \mathcal{X}$ is a collision-resistant function, then $H_{MD}: \{0,1\}^{\leq L} \Rightarrow \mathcal{X}$ is a CRHF
- Let there exist a PPT $\mathcal{A}_{\text{MD}}$ which outputs distinct $M, M' \in \{0,1\}^{\leq L}$, such that:

$$\Pr[H_{MD}(M) = H_{MD}(M')] = f(n), \text{where } f(n) \text{ is a non-negligible function}$$

$\hat{M}$
m_1
m_2
...
m_u
$t_0 = \text{IV}$
h
t_1
t_2
t_{u-1}
$t_u := H_{MD}(M)$
$\widehat{M'}$
m'_1
m'_2
m'_v
t'_1
t'_2
t'_{v-1}
$t'_v := H_{MD}(M')$

- Using $\mathcal{A}_{\text{MD}}$, we construct a PPT $\mathcal{A}_h$ which outputs distinct pairs $\underline{(t,m),(t^*,m^*)} \in \mathcal{X} \times \mathcal{Y}$:

$$\Pr[h(t,m) = h(t^*,m^*)] = f(n)$$

- To find the collision $(t,m),(t^*,m^*)$ for h, $\mathcal{A}_h$ parses the hash-chain $H_{MD}(M)$ and $H_{MD}(M')$ from right to left

So, now let us see whether indeed applying this Merkle-Damgard paradigm iteratively over a fixed-length collision-resistant compression function ends up giving you a collision-resistant hash function or not. So we are going to prove that that if indeed the function h is a fixed-length compression function and as well as collision-resistant, then by applying the Merkle-Damgard paradigm, the function H which we have obtained which can hash any bit string of length up to L bits indeed is a collision-resistant function.

The proof will be by reduction or by a contradiction, namely we can prove that assume if you have a poly-time adversary, which I denote as A_{MD}, which when given the description of the

Merkle-Damgard paradigm and a description of your fixed-length compression function outputs a pair of distinct messages or a collision for the bigger function or the function H that we have constructed. That means, the algorithm A_{MD} outputs a pair M, M', where M and M' are distinct, but still the hash value of M and M' operated with respect to the H function is same.

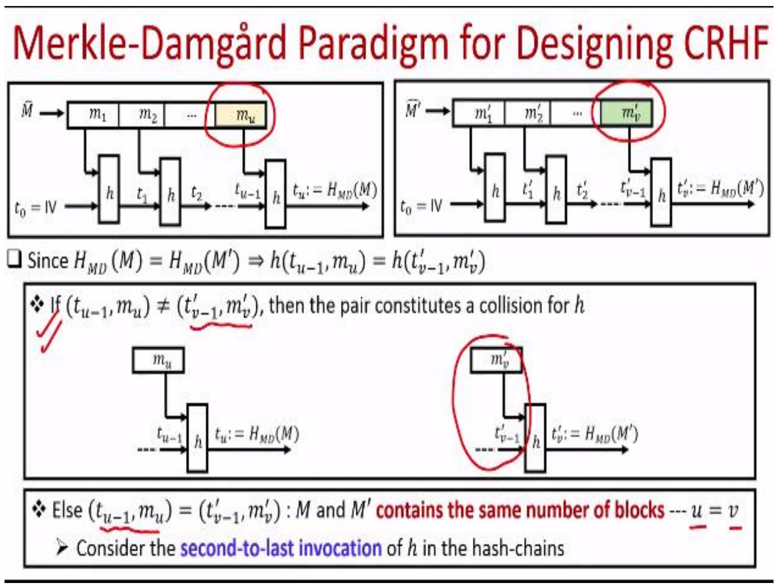
The probability of collision here is say f(n) where f(n) is a non-negligible function. So what we are going to prove is that if we have such an adversary A_{MD}, then using this adversary A_{MD} we can construct or we show how to construct another poly-time adversary A_h, which can give you a collision for your fixed-length collision-resistant compression function h with the same probability with which the adversary A_{MD} could have given you a collision for the function H_{MD}.

Basically, the idea of this adversary A_h is that the adversary A_h, its goal is to find out a collision for the h function, namely its goal is to come up with a pair t, m and t*, m* such that the output of this h function on these two inputs is the same and to find is the pair t, m and t*, m*, what your adversary A_h does is basically it parses the sequence of hash chain which your Merkle-Damgard paradigm would have created while hashing this message M and for hashing the message M', where the message M and M' are produced by the adversary is A_{MD}.

So pictorially, this is how the hash value for the message M would have been computed as per the Merkle-Damgard paradigm, right. The message M would have been first converted into an encoded input and then we would have applied the function h iteratively and in the same way to compute the value of the function H_{MD} on the message M', we would have applied the function h iteratively as this. We would have first converted the input M' into an encoded input and then we would have applied the function H iteratively and would have obtained the value of hash value on the message M'.

So, what basically the adversary A_h is going to do is it is going to compare this two hash chains, the hash chain on the left side and the hash chain on right side and our claim is that if m, m' constitutes a collision where m and m' are distinct, then definitely there will be at least one collision present at some place in the hash chain of M and hash chain of M', right.

(Refer Slide Time: 23:18)



Merkle-Damgård Paradigm for Designing CRHF

❑ Since $H_{MD}(M) = H_{MD}(M') \Rightarrow h(t_{u-1}, m_u) = h(t'_{v-1}, m'_v)$

❖ If $(t_{u-1}, m_u) \neq (t'_{v-1}, m'_v)$, then the pair constitutes a collision for h

❖ Else $(t_{u-1}, m_u) = (t'_{v-1}, m'_v)$: M and M' **contains the same number of blocks** --- $u = v$

➢ Consider the **second-to-last invocation** of h in the hash-chains

So let us see how exactly the collision for the function h is going to be obtained based on these two hash chains. So since the hash value of the message M and the hash value of the message M’ are same, right, because that is a collision for your function, H_{MD}. So what we are going to do is we are going to focus on the last invocation of the h function in the hash chain for the message M and the last invocation of the h function in the hash chain for the message M’, right.

So the last invocation for the message M is on your left hand side and the last invocation for the hash chain on M’ is on the right hand side and as you can see, you have the inputs m_u and t_{u-1} for the hash chain for M and the input for the hash chain for M’ is the input m'_v and t'_{v-1}. So, there could be 2 possibilities. If this joint input t_{u-1} concatenated with m_u is different from t'_{v-1} concatenated with m'_v, then that itself creates a collision for your h function.

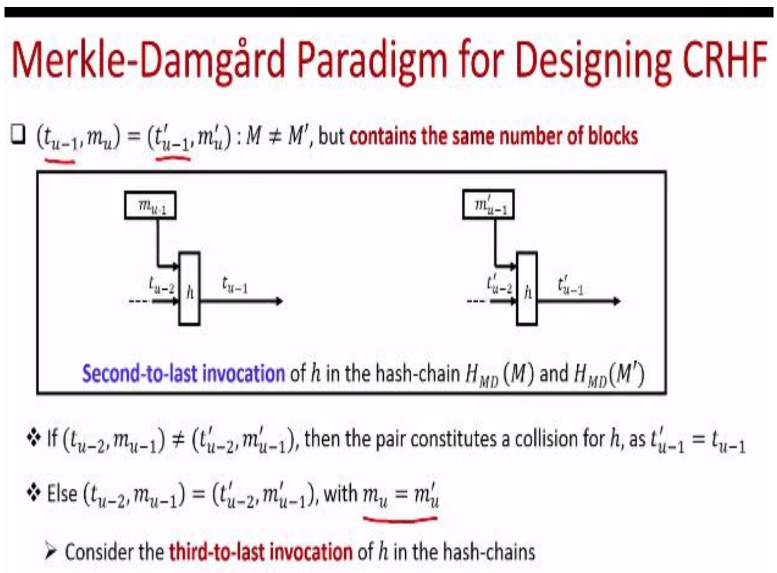
Because what is happening here is that even though the combined input here is different from the combined input here, their output with respect to the h function are same because that is what is the overall output of the hash value H for the message M and M’. So, if we are in this case, then we have spotted a collision very easily, namely we have spotted the collision with respect to the last invocation of the h function.

Otherwise, it implies that the combined input t_{u-1} and m_u is the same as the combined input t'_{v-1} and m'_v and that automatically means that the number of blocks which are present in the message M and M’ are same, namely u = v because we are in the case when this m_u and m'_v

are same, right, and remember m_u and m'$_v$ are going to consist of some number of bits of the actual message and padded bits, though combinely they are the same.

That means, the number of blocks of *l* bits which are present in the message M and a number of blocks of *l* bits which are present in the message M' are actually same. So, if you are in the else case that means we have the last invocation of the h function in the hash chain of M and the hash of M' does not constitute a collision.

(Refer Slide Time: 26:08)



Merkle-Damgård Paradigm for Designing CRHF

- ❑ $(t_{u-1}, m_u) = (t'_{u-1}, m'_u)$: $M \neq M'$, but contains the same number of blocks

m_{u-1}
t_{u-2}
h
t_{u-1}
m'_{u-1}
t'_{u-2}
h
t'_{u-1}
Second-to-last invocation of h in the hash-chain $H_{MD}(M)$ and $H_{MD}(M')$

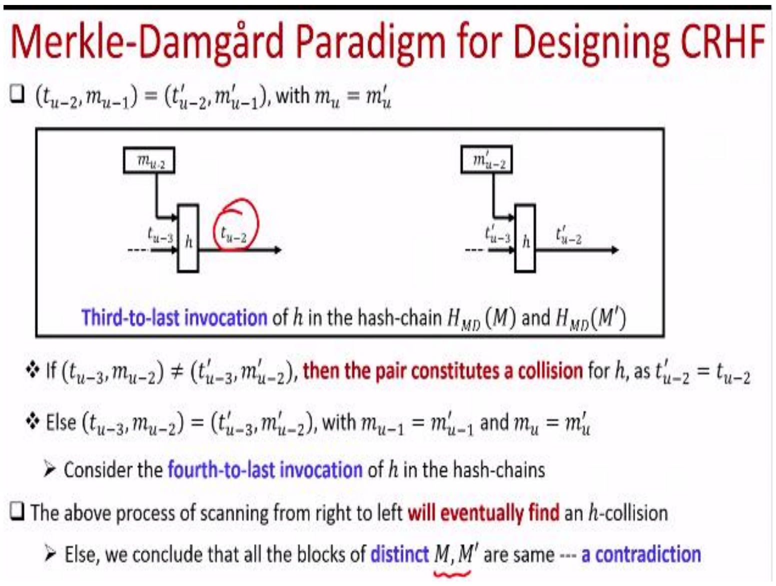
- ❖ If $(t_{u-2}, m_{u-1}) \neq (t'_{u-2}, m'_{u-1})$, then the pair constitutes a collision for h, as $t'_{u-1} = t_{u-1}$
- ❖ Else $(t_{u-2}, m_{u-1}) = (t'_{u-2}, m'_{u-1})$, with $m_u = m'_u$
  - ➢ Consider the third-to-last invocation of h in the hash-chains

So what we have to do is now we have to go one step back in the hash chain in the respective hash chains and focus on the second-to-last invocation of the h function in the respective hash chains, right. So, if we continue in the else case, we are here. We have distinct messages M and M' and now we are focusing on the second-to-last invocation of the h function in their respective hash chains. Now, again we have 2 possible cases.

If the joint input to the h function in the respective hash chains are different, then we have spotted a collision for the h function because now we are in the case where even though the combined input to the second-to-last invocation of the h function in the respective hash chains are different, their outputs are same, right? Because the outputs are t_{u-1} and t'$_{u-1}$ which we already know they are same because we are under that case.

But again, if it so happens that the combined input for the second-to-last invocation of the h functions in the respective hash chains are equal, right, then it gives you the implication that else m_u and m_u' are same and combined input are same, then we have to go one step back in the respective hash chains and we have to focus on the third-to-last invocation of the h function in the respective hash chains.

(Refer Slide Time: 27:35)



Merkle-Damgård Paradigm for Designing CRHF

- ❑ $(t_{u-2}, m_{u-1}) = (t'_{u-2}, m'_{u-1})$, with $m_u = m'_u$

m_{u-2}
t_{u-3}
h
t_{u-2}
m'_{u-2}
t'_{u-3}
h
t'_{u-2}
Third-to-last invocation of h in the hash-chain $H_{MD}(M)$ and $H_{MD}(M')$

- ❖ If $(t_{u-3}, m_{u-2}) \neq (t'_{u-3}, m'_{u-2})$, then the pair constitutes a collision for h, as $t'_{u-2} = t_{u-2}$
- ❖ Else $(t_{u-3}, m_{u-2}) = (t'_{u-3}, m'_{u-2})$, with $m_{u-1} = m'_{u-1}$ and $m_u = m'_u$
  - ➢ Consider the fourth-to-last invocation of h in the hash-chains
- ❑ The above process of scanning from right to left will eventually find an h-collision
  - ➢ Else, we conclude that all the blocks of distinct M, M' are same --- a contradiction

So, again if you now go to the third-to-last invocation of the h function in the respective hash chains, we are in this condition and again we have to argue whether the combined input to the h function in the respective hash chains are same or not. If they are not, then this instance of h basically creates a collision for the h function where we have a pair of inputs which are different, but the output namely t'_{u-2} and t_{u-2} are same, but if not, then again we have to go to the previous invocation of the h function in the respective hash chains and so on.

So, what basically, we are doing here is we are parsing the hash chain of the M and M' respectively and the claim is that eventually we will end up finding a collision because if we do not find collision and if we end up parsing from right to left and come to the beginning of the message, first block of M and the first block of M', and it means that even though we have distinct M and M', all the blocks of M and M' are same, which is going to be a contradiction.

But since M and M' are different because that constitutes a collision for your overall hash function H_{MD}, definitely there will be a collision which will be spotted with respect to the h function. So, that proves that if indeed our h function is a fixed-length compression function and collision resistant, then by applying the Merkle-Damgard paradigm, the overall function which we are going to obtain is also collision resistant. That means in poly time, it would not be possible to come up with a pair of distinct inputs, which gives you the same hash value.

So, that brings me to the end of this lecture. Now, just to summarize, in this lecture, we started discussing about interesting cryptography primitive called cryptographic hash function. We have seen the definition of collision resistance with respect to the cryptographic hash functions. Of course, there are several other properties for cryptographic hash functions like preimage resistance and second preimage resistance, but we are not going to discuss those properties because we would not be using them in this course.

We also discussed about the Merkle-Damgard paradigm which is a well known paradigm used in several practical instantiations of cryptographic hash function and what basically Merkle-Damgard paradigm does is it takes any fixed-length collision resistant compression function, apply a collision-resistant hash function for any size input. Thank you.