Cryptography and Network Security

Prof. D. Mukhopadhyay

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

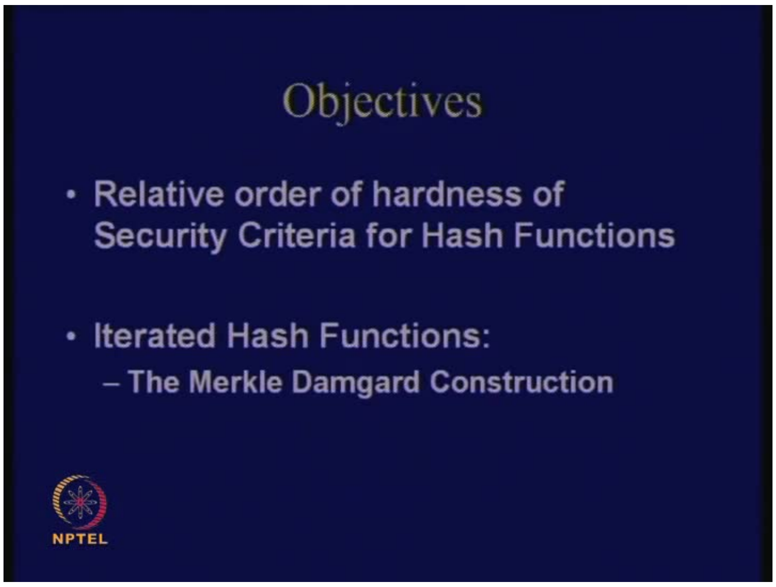
Module No. # 01

Lecture No. # 24

Cryptographic Hash Functions (Contd.)

We shall continue with the topic of cryptographic hash functions, which we started last day.

(Refer Slide Time: 00:25)

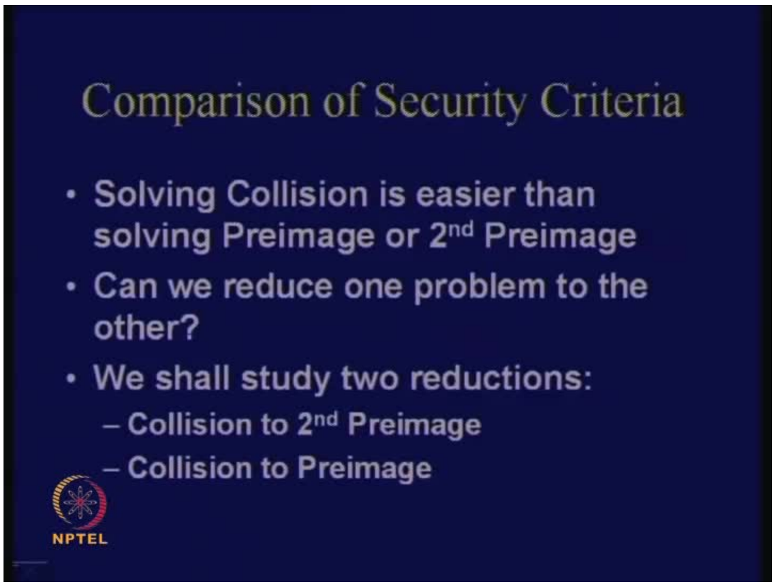


Objectives

- Relative order of hardness of Security Criteria for Hash Functions
- Iterated Hash Functions:
  - The Merkle Damgard Construction

We were essentially discussing about the relative order of hardness of security criteria for hash functions, we will continue with that. Then, discuss about a particular type of construction, which is very much popular, it is called the Merkle Damgard construction. It is an integrate construction technique for hash functions, so we will discuss about these topics.

(Refer Slide Time: 00:44)

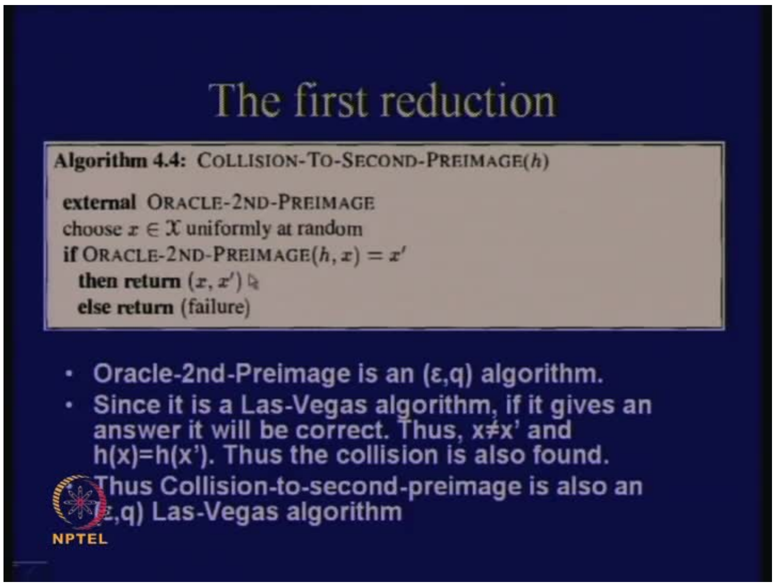


Comparison of Security Criteria

- Solving Collision is easier than solving Preimage or 2nd Preimage
- Can we reduce one problem to the other?
- We shall study two reductions:
  - Collision to 2nd Preimage
  - Collision to Preimage

To start with, we have discussed about these two reductions. We were discussing about the collision to second preimage problem; that we discussed last day. We will be discussing about the collision to preimage problem.

(Refer Slide Time: 01:00)



The first reduction

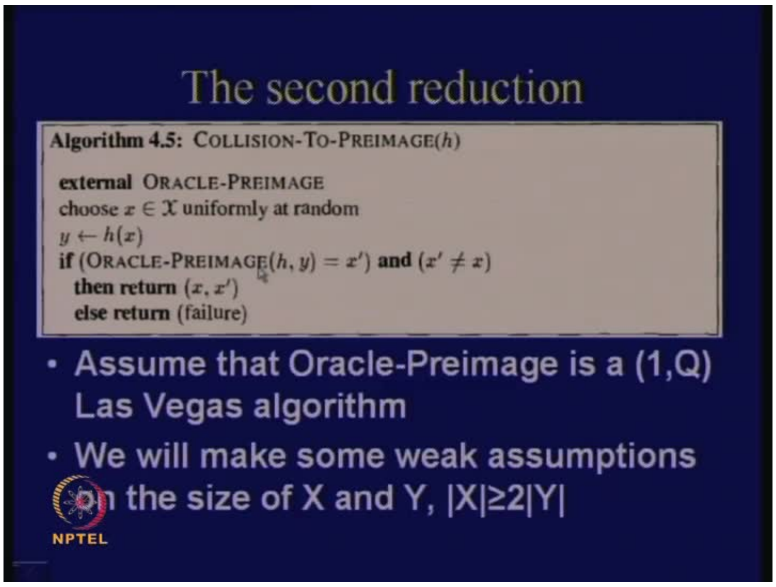
```
Algorithm 4.4: COLLISION-TO-SECOND-PREIMAGE($h$)
 external ORACLE-2ND-PREIMAGE
 choose $x \in \mathcal{X}$ uniformly at random
 if ORACLE-2ND-PREIMAGE($h, x$) $= x'$
   then return $(x, x')$
   else return (failure)
```

- Oracle-2nd-Preimage is an (ε,q) algorithm.
- Since it is a Las-Vegas algorithm, if it gives an answer it will be correct. Thus, x≠x' and h(x)=h(x'). Thus the collision is also found.
- Thus Collision-to-second-preimage is also an (ε,q) Las-Vegas algorithm

One thing actually is we were considering these types of reductions, there were some queries like why do we put this? One reason is, because this is also a probabilistic algorithm, therefore this does not always give you the answer; it gives you the answer sometimes, it does not give you the answer sometimes. For example, this is a Las-Vegas randomized algorithm, therefore this can fail also. When this algorithm fails, then this collision to second preimage also fails.

Therefore, this is only to check whether this actually gives you an answer or not. Therefore, that means that in case of a Las-Vegas algorithm, whether it terminates or not, there is a probability of it terminating and that probability is denoted by epsilon. So, if that is the probability, then the probability with this collision to second preimage also gives you a correct answer, is also epsilon. That is the precise design why this - if was kept actually, but again as we discussed in the last day's class, we do not require to check for x and x dash being equal or not, because since this is correctly giving an answer for the second preimage problem, therefore this is automatically taken care of.

(Refer Slide Time: 02:18)



The second reduction

```
Algorithm 4.5: COLLISION-TO-PREIMAGE($h$)
  external ORACLE-PREIMAGE
  choose $x \in \mathcal{X}$ uniformly at random
  $y \leftarrow h(x)$
  if (ORACLE-PREIMAGE($h, y$) $= x'$) and ($x' \neq x$)
    then return ($x, x'$)
    else return (failure)
```

- Assume that Oracle-Preimage is a (1,Q) Las Vegas algorithm
- We will make some weak assumptions on the size of X and Y, |X|≥2|Y|

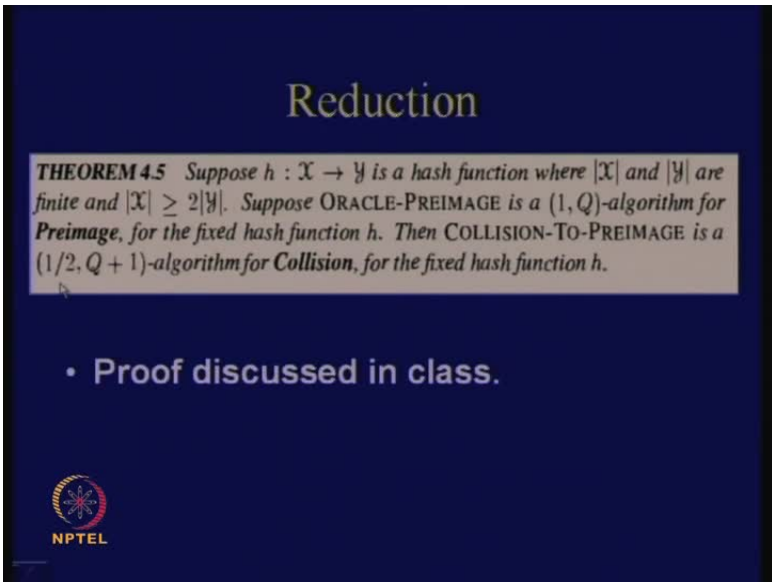
This is something we have discussed; we were discussing about this particular reduction. Therefore, this is the collision to preimage problem. We know that preimage problem, means given h x, we have to return an x value for which the hash value are the same; that is the preimage problem.

We assume that there is a solution to this preimage problem; from there we show that in that case, we can also solve the collision problem. Therefore, if this is the question that is if you are choosing x uniformly at random, then you compute the hash value that is h x value and then engage the algorithm for computing the preimage. Therefore, this is an oracle to which you are asking a question and it gives you back an x dash.

One thing you have to now check is, whether x and x dash are same or not, because this just gives you the preimage. Therefore, you do this check that is whether x dash and x are same or not, if they are not same, then you can return x comma x dash, because both of them have the same hash value and they are also not equal. Therefore, this is the solution to your collision problem as well.

Now, the question is that if I assume that the oracle preimage is a 1 comma Q Las-Vegas algorithm, so that means 1 comma Q, means the probability of giving you the preimage solution is actually 1. There are Q queries required, then what is the corresponding epsilon values and number of queries required from this collision to preimage to occur.

(Refer Slide Time: 04:00)



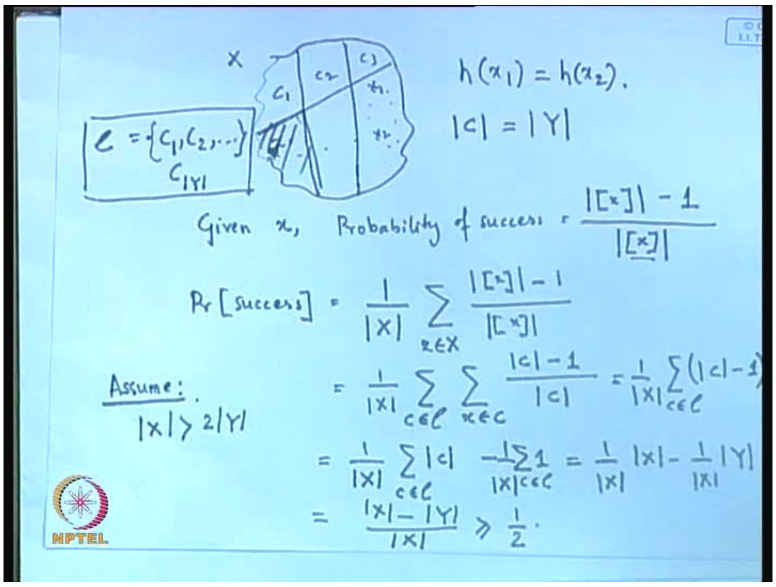
Reduction

THEOREM 4.5 Suppose $h : \mathcal{X} \rightarrow \mathcal{Y}$ is a hash function where $|\mathcal{X}|$ and $|\mathcal{Y}|$ are finite and $|\mathcal{X}| \geq 2|\mathcal{Y}|$. Suppose ORACLE-PREIMAGE *is a $(1, Q)$-algorithm for* **Preimage**, *for the fixed hash function h. Then* COLLISION-TO-PREIMAGE *is a $(1/2, Q+1)$-algorithm for* **Collision**, *for the fixed hash function h.*

- Proof discussed in class.

Number of queries you can easily see that if this query is Q, since there is an extra query, it is Q plus 1. About the probability we will make an assumption, so the assumption is as follows that is the cardinality of x is twice greater than the cardinality of y. Then, we can show that you have actually the probability of half comma Q plus 1. In order to understand that let us consider a proof.

(Refer Slide Time: 04:21)



X

$\boxed{\mathcal{C} = \{C_1, C_2, \ldots, C_{|Y|}\}}$

$h(x_1) = h(x_2).$

$|\mathcal{C}| = |Y|$

Given x, Probability of success $= \dfrac{|[x]| - 1}{|[x]|}$

$$\Pr[\text{success}] = \frac{1}{|X|} \sum_{x \in X} \frac{|[x]| - 1}{|[x]|}$$

Assume: $|X| \geq 2|Y|$

$$= \frac{1}{|X|} \sum_{c \in \mathcal{C}} \sum_{x \in c} \frac{|c| - 1}{|c|} = \frac{1}{|X|} \sum_{c \in \mathcal{C}} (|c| - 1)$$

$$= \frac{1}{|X|} \sum_{c \in \mathcal{C}} |c| - \frac{1}{|X|} \sum_{c \in \mathcal{C}} 1 = \frac{1}{|X|} |X| - \frac{1}{|X|} |Y|$$

$$= \frac{|X| - |Y|}{|X|} \geq \frac{1}{2}.$$

The proof works as like this, so consider the space of x. Now, what we start doing is that we start partitioning this space, so we start inducing partitions; you know that we can induce partitions by equivalence relations. Therefore, we can start inducing partitions; you start partitioning this set such that all the values which lie in one partition they have got the same hash value. That means if there are two values like x 1 and x 2, which is one partition and then h of x 1 and h of x 2 are the same. That is the definition of a partition in this case.

Let us now consider that. Suppose - how many such partitions are there? Therefore, the number of partitions which will be there, suppose all of them are some collections, therefore this is my c 1, this is c 2, this is c 3 and so on, then the number of such collections will be equal to the cardinality of y, because all of the partitions are indicating one particular hash value. Therefore, the number of collections is actually equal to the cardinality of y.

Now, let us consider that there is a given value of x; suppose you have been provided with an x value, what is the probability of success with which you are actually solving the collision problem? Your probability of success will be actually equal to - how many total number of given x? How many total values are there? There are actually - if I call this as - I mean the equivalence class - I mean, for there is basically one partition, if I indicate by this symbol, then the cardinality of this is the total number of possible values, which I can choose from among them. Except one, all of them are my correct values, because one will be the same value. So, do you understand what I am saying?

Suppose, let us consider one particular partition. You are considering this partition and suppose, x lies in this partition. In that case, when you are asking the preimage to give you a result, the preimage is giving you a result with a probability of 1, the preimage oracle, we are assumed that it has - it is a Las-Vegas algorithm with probability 1. Therefore, it will definitely solve the preimage problem and it will return you some value.

What are possible values which it can return? It can return the number of values which are lying in this partition. What is that set? So, I indicate the number by equivalence of x and the cardinality of that. I mean, in that except for the number x itself or the value x itself, all are correct values for the collision problem. Therefore, your probability of success, if you are given this value of x, is this; any doubts?

Now, we need to consider the average case probability. What we will do? If I am interested in the average probability, then what I will do is that I will take it over the entire set. Therefore, I will do a sigma operation of this and I will value x over the entire set x.

This, we can actually write in this fashion. So, there are so many collections, let us break this sigma into two sigmas. In this particular sigma, let us concentrate that x is belonging to one particular collection. Here, the collection belongs to all possible collection sets. Let c be the collection sets, which comprises of c 1, c 2 and so on. How many total collections are there? c cardinality of y.

Therefore, consider that this is your total collection set; therefore, you know that this is how what you are doing. What does these value compute to, this is the number of elements in the corresponding collection c; therefore this is also cardinality of c minus 1. This will compute to now one by cardinality of x, if I keep this sigma, so this will be for all the values of x, which lies in cardinality of c. So, how many values are there? There is cardinality of c values, this value’s a constant and you get cardinality of c minus 1. This computes to 1 by x, I can break this into cardinality of c, where c lies to this cardinality of set minus sigma 1, this also is this, is denominator working, therefore the c is again varying over this cardinality set of c (Refer Slide Time: 09:50).

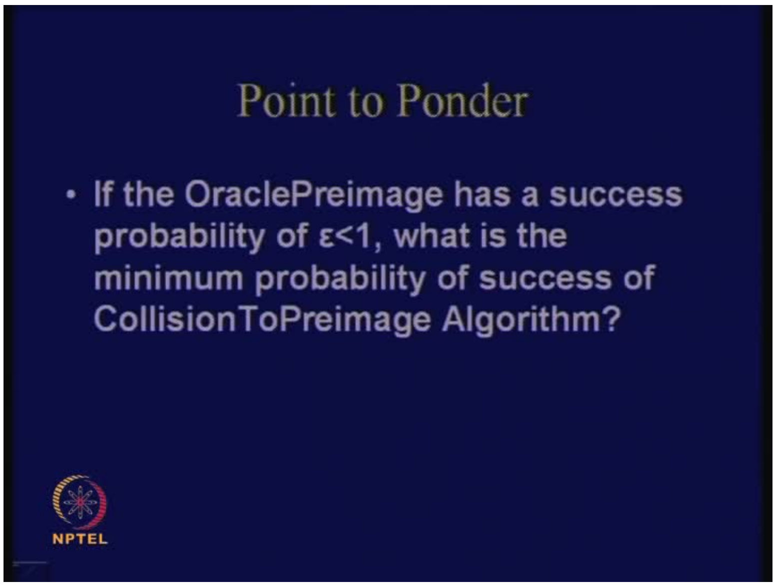
Therefore, what does the first term compute to? It computes to mod x. What does the second term compute to? Mod y, because for each collection you are getting a 1, so this computes to mod y. Therefore, this is nothing but mod of x minus mod of y divided by mod of x.

Now, you see that we have actually made one assumption in the theorem, which says that mod of x is greater than twice of mod of y. If I assume this, this is quite a practical assumption to make, in that case, mod of x is greater than twice of mod of y, if I assume this and then this particular thing is greater than half. Therefore, your probability of success is at least equal to half.

You note one thing that whenever we are doing this kind of reductions or the reduction that we have discussed in our class, we have assumed an ideal hash function assumption - We have assumed that the hash functions are ideal. Basically, all these proofs that we have shown are under the random oracle model. So, if the random oracle model is violated, then these proofs or these reductions may not hold true; so we have to be careful.

What was the ideal hash function model? The main assumption was that if I need to compute a new hash value, how many previous hash values you have computed, if you are computing the hash for new input, then you have to again compute it, previous value should not help you; very informally this was the meaning.

(Refer Slide Time: 12:23)

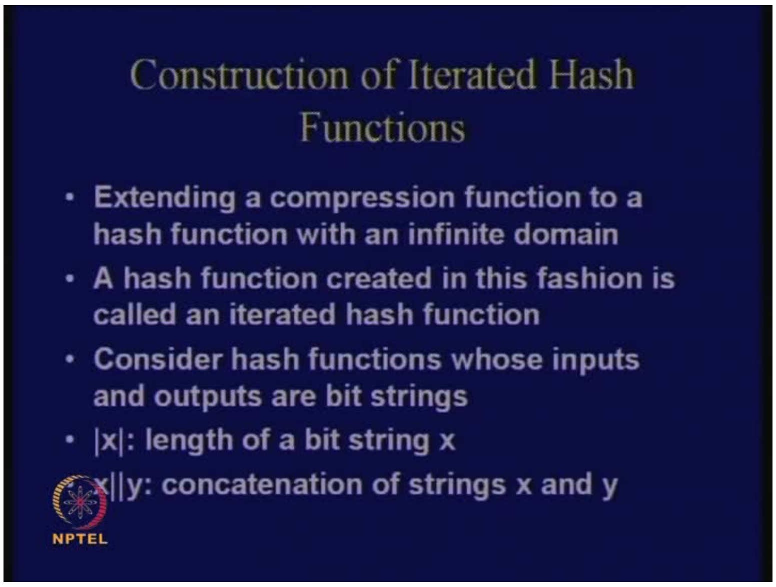


Point to Ponder

- If the OraclePreimage has a success probability of $\varepsilon<1$, what is the minimum probability of success of CollisionToPreimage Algorithm?

Now, you can actually ponder upon a point, it says that if the oracle preimage has a success probability of epsilon which is less than 1, then what is the minimum probability of success of collision to preimage algorithm? You can just think on this actually.

(Refer Slide Time: 12:43)



Construction of Iterated Hash Functions

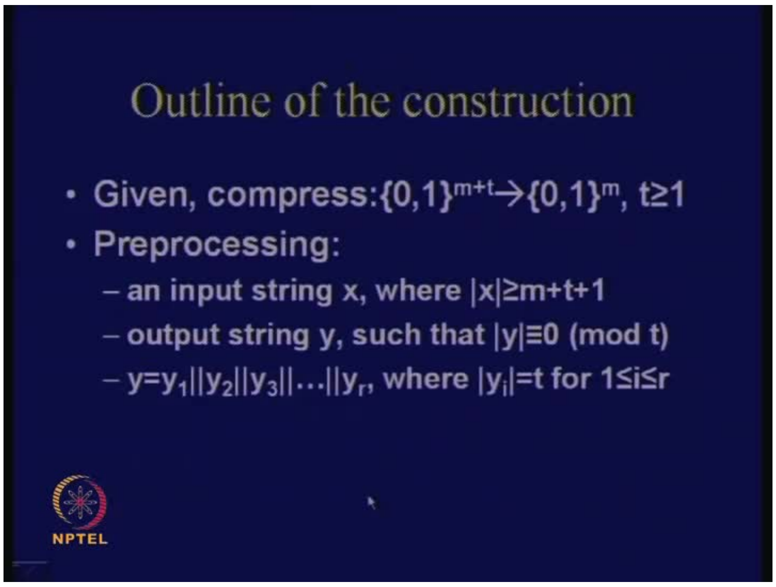
- Extending a compression function to a hash function with an infinite domain
- A hash function created in this fashion is called an iterated hash function
- Consider hash functions whose inputs and outputs are bit strings
- |x|: length of a bit string x
- x||y: concatenation of strings x and y

Now, we go into the construction of iterated hash functions. We will take up a particular type of construction, which is known as the Merkle Damgard construction. The idea is that we have got a compression function. So, all these iterated hash functions has got a underlying concept that is, there is a compression function, which means that it takes a large number of values and it compresses to a small output bits. Using this, we will iterate this, so that my domain actually becomes infinite and my output is still constant.

Now, if you see this slide, extending a compression function to a hash function with an infinite domain, so that is the objective of the iterated algorithm. A hash function is created in this fashion is called an iterated hash function. So, we will consider a hash function whose inputs and outputs are bit strings, which means they have 0 1 values.

Again, we know this that mod x denotes the length of a bit string x and this particular symbol is a concatenation symbol. So, x concatenated y will be represented in this fashion, so x is a string, y is a string and they concatenate in this form. So, these are some notations.

(Refer Slide Time: 14:02)

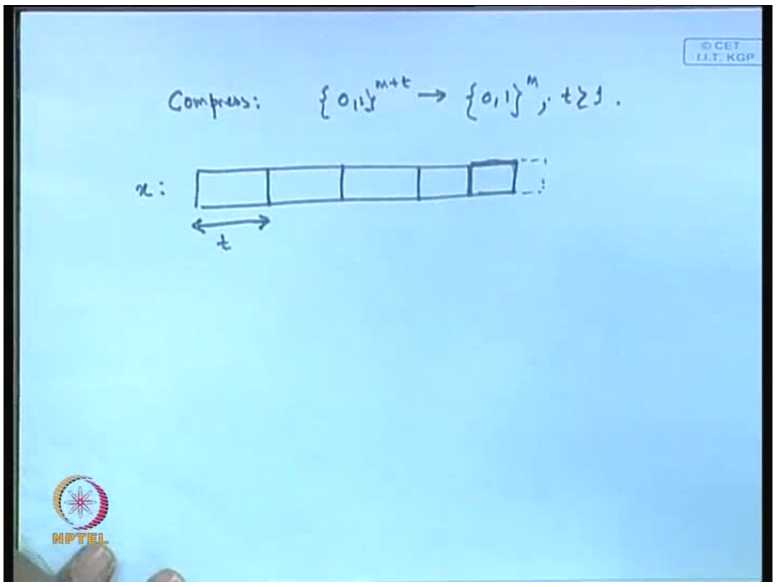


Outline of the construction

- Given, compress:$\{0,1\}^{m+t} \to \{0,1\}^m$, $t \geq 1$
- Preprocessing:
  - an input string x, where $|x| \geq m+t+1$
  - output string y, such that $|y| \equiv 0 \pmod{t}$
  - $y = y_1 || y_2 || y_3 || \ldots || y_r$, where $|y_i| = t$ for $1 \leq i \leq r$

Now, let us start the algorithm, therefore the basic concept behind this construction is that you have got a compress function, which means that you have got a compress block, which will take in say an m plus t bits and will give you a 0 1 m bit output.

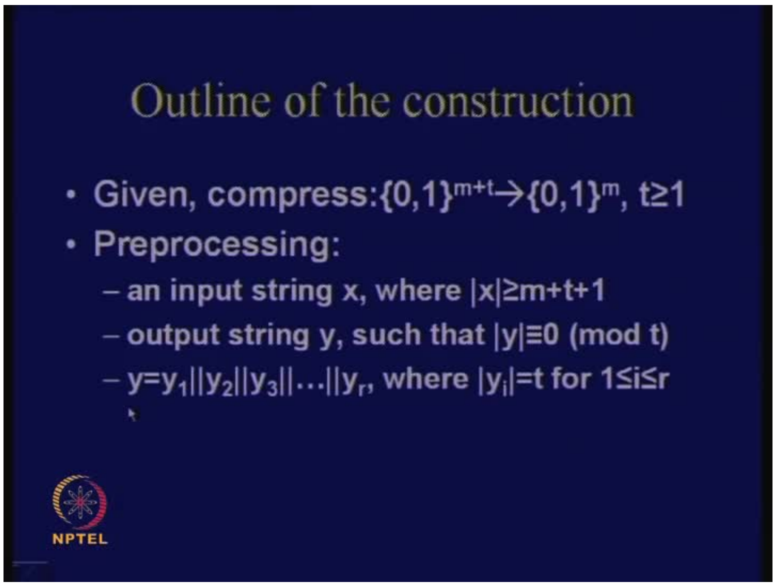
(Refer Slide Time: 14:16)



Therefore, this is a compression function, so t is typically greater than equal to 1. Now, what will we do is that your input x can be actually quite large, using that you are computing the hash value. The first thing which you do is that you have got the input string x, you can think that you can actually break this - break them into blocks. What you can do is that you can break them into sub blocks, such that each block is actually divisible by t or rather is equal to t.

Therefore, you understand that in the last block, there can be some values - I mean it is not exactly equal to t. Therefore, what you do is that you do padding. You extend this and you make it also equal to t. This particular step is actually called the preprocessing step. Therefore, what you do is that you take x and you make the output of this particular preprocessing step, the size of that block, a multiple of t.

(Refer Slide Time: 15:48)

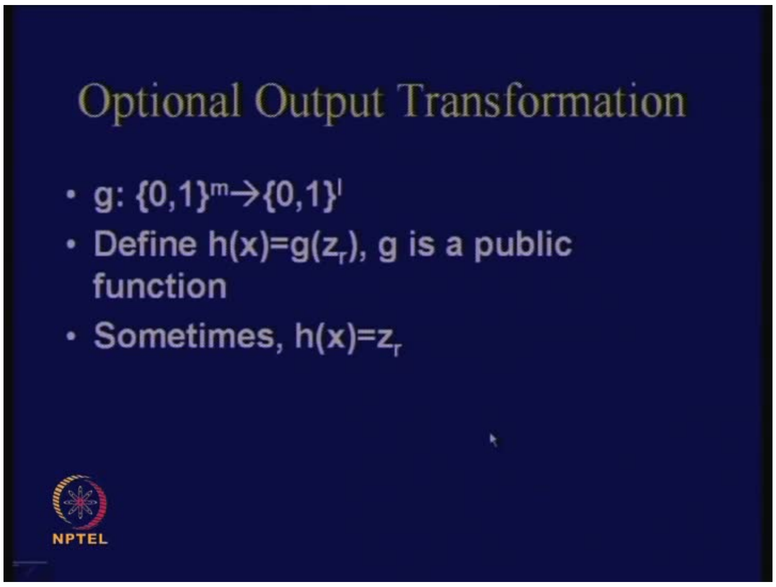


Outline of the construction

- Given, compress:$\{0,1\}^{m+t} \rightarrow \{0,1\}^m$, $t \geq 1$
- Preprocessing:
  - an input string x, where $|x| \geq m+t+1$
  - output string y, such that $|y| \equiv 0 \pmod t$
  - $y=y_1||y_2||y_3||\dots||y_r$, where $|y_i|=t$ for $1 \leq i \leq r$

Therefore, you see that what you do is that you take an input string x, where mod x is greater than equal to m plus t plus 1. The output string y is such that mod of y is actually equal to 0 mod t, which means it is divisible by t and y; you are actually breaking up like y 1, y 2 and so on, till y r. Therefore, r - I mean, for each block, is actually equal to t. So that means r into t should be the cardinality of y. So, cardinality of y means the number of bits, which are there in the block y.

(Refer Slide Time: 16:18)

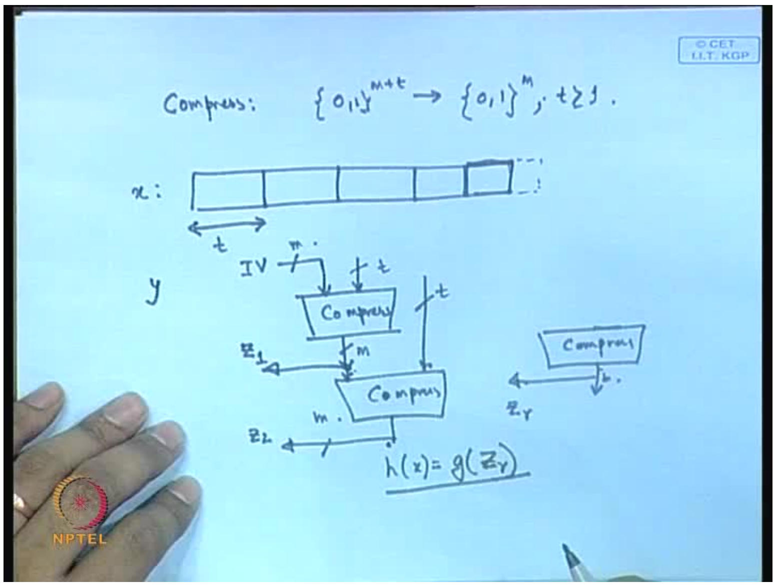


Optional Output Transformation

- g: $\{0,1\}^m \rightarrow \{0,1\}^l$
- Define $h(x)=g(z_r)$, g is a public function
- Sometimes, $h(x)=z_r$

Then, you have got certain steps; I mean you do a sort of an iterated algorithm. Therefore, what you do is that you take one steps that is you start computing with y; now you got this y. So, this y has got each of the blocks as size of t.

(Refer Slide Time: 16:36)



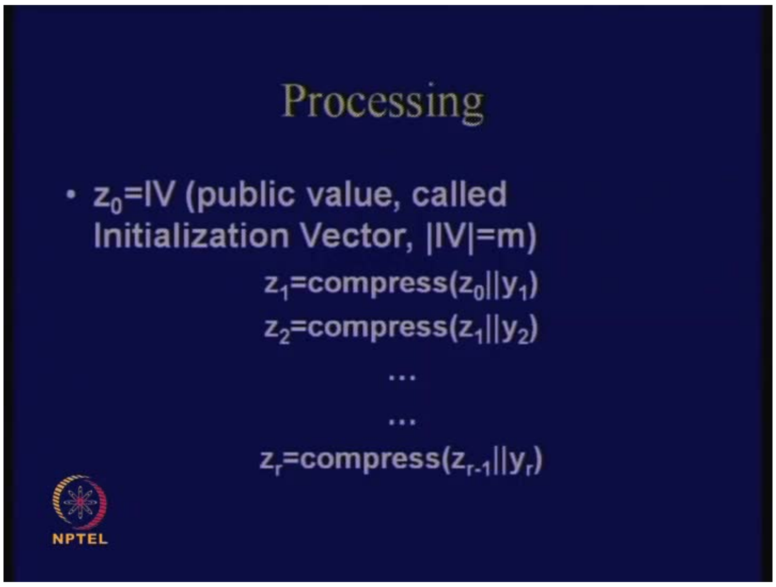
You have got the compress function, which takes m plus t bits and gives you m bit output. What you do is that now to this compress function, you feed in t bits from y and you feed in m bits, so you start with something which I called as an IV or an initializing vector.

Now, this will result in an m bit output. This m bit output, what I am doing is that I am passing on as my first as some parameter, which means, suppose I can call that as Z 1. Next, what I do is that I take this m bit and I take the next t bits from y, again I pass that to my compress function, I again obtained another m bits, I call that Z 2. Similarly, I keep on repeating this process. The final compress steps which you obtain also will therefore result in an m bit output.

This particular output, let us write that as Z r, because there are r blocks of y, this Z r is sometimes called the hash function output or Z r is often given an optional transformation like for example, g Z r and that is actually called h x. So, g is optional, which means that you can either make it an identity function or you can have a g transformation.

((Student Talks))

(Refer Slide Time: 19:15)



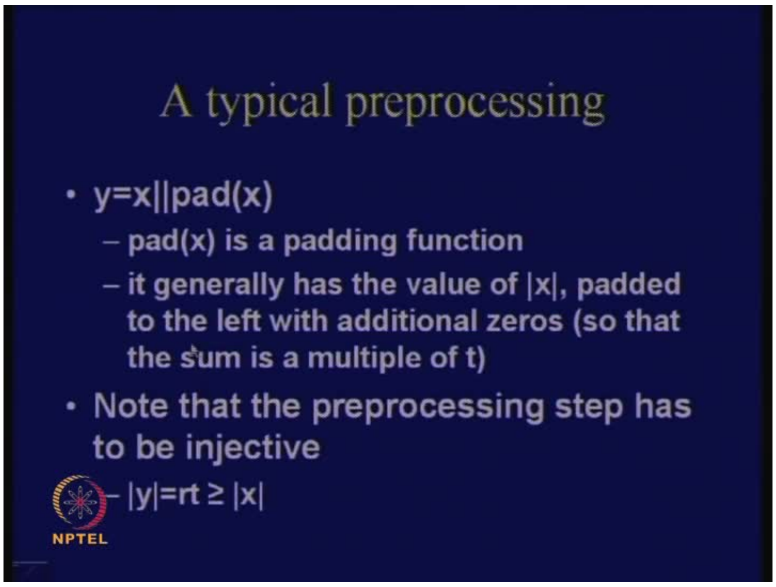
Processing

- z_0=IV (public value, called Initialization Vector, |IV|=m)

z_1=compress($z_0||y_1$)
z_2=compress($z_1||y_2$)
...
...
z_r=compress($z_{r-1}||y_r$)

This m bit is actually fed to the compress, therefore this compress now has got an m plus t bits of input and it also results in an m bit output. This is the way you keep on iterating this process. Therefore, this final Z r which you obtained, like you are exporting some values like Z 1, Z 2 and so on, thus rth value that is Z r, is sometimes given an optional transformation, sometimes not; optional means sometimes you do not give it. Therefore, g Z r is actually called h x; this is the basic idea behind iterated hash functions.

(Refer Slide Time: 19:22)



A typical preprocessing

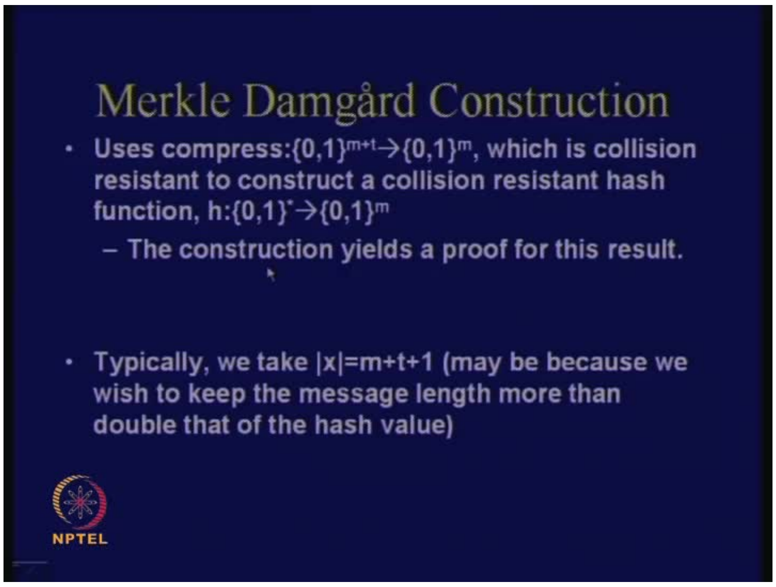
- y=x||pad(x)
  - pad(x) is a padding function
  - it generally has the value of |x|, padded to the left with additional zeros (so that the sum is a multiple of t)
- Note that the preprocessing step has to be injective
  - |y|=rt ≥ |x|

This is what I have already told to you. So, you can go back looking into the slide, but so let us consider one typical preprocessing step. Therefore, what it does is that if you see that you have got y and you have got x, what you do is that you concatenate it with pad x. So, pad x is nothing but a simple padding function, so it generally has the value of mod x. What you do is that you pad it to the left with additional zeros. So that the sum is a multiple of t, so this is a very simpler step which I have already told you.

Now, you note one thing that I mean the main thing is that to be noted here is that this preprocessing step has to be an injective function. What does injective function mean? It has to be a one to one transformation, why? Because, if it is not a one to one transformation, then again concentrate on this preprocessing step itself and create a collision. So, collision creating should be quite easy in that case. Therefore, you have to ensure that the preprocessing step is actually a one to one mapping.

Therefore, you note that the preprocessing step has to be injective and you see that mod of y that is the number of values in y, is actually equal to r t, which is greater than the value of number of values in x. This is necessary for one to one transformation, this cannot be lesser than mod of x. So, these are some minor points.

(Refer Slide Time: 20:44)

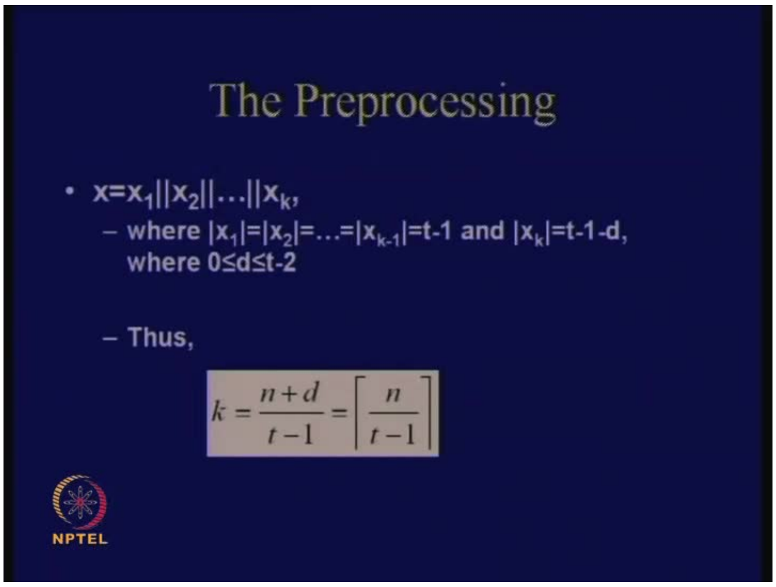


Merkle Damgård Construction

- Uses compress:$\{0,1\}^{m+t} \rightarrow \{0,1\}^m$, which is collision resistant to construct a collision resistant hash function, h:$\{0,1\}^* \rightarrow \{0,1\}^m$
  - The construction yields a proof for this result.
- Typically, we take $|x|=m+t+1$ (may be because we wish to keep the message length more than double that of the hash value)

Now, we coming to the original Markle Damgard construction, let us consider upon this. This basically the same iterated hash function technique only, but slightly there are some small minor changes, so let us concentrate. What you do is that again you use the compress function, which is an m plus t bit to m bits. So, this is a collision resistant, this is used to construct a collision resistant hash function. Therefore, you see that the domain of your hash function is actually an arbitrary size 0 1 string, from there you are obtaining a 0 1 m bit output. The main advantage of using or main elegance of using a Markle Damgard construction is that it gives you a wonderful proof.

(Refer Slide Time: 21:31)



The Preprocessing

- $x=x_1||x_2||\ldots||x_k$,
  - where $|x_1|=|x_2|=\ldots=|x_{k-1}|=t-1$ and $|x_k|=t-1-d$, where $0 \leq d \leq t-2$
  - Thus,

$$k=\frac{n+d}{t-1}=\left\lceil\frac{n}{t-1}\right\rceil$$

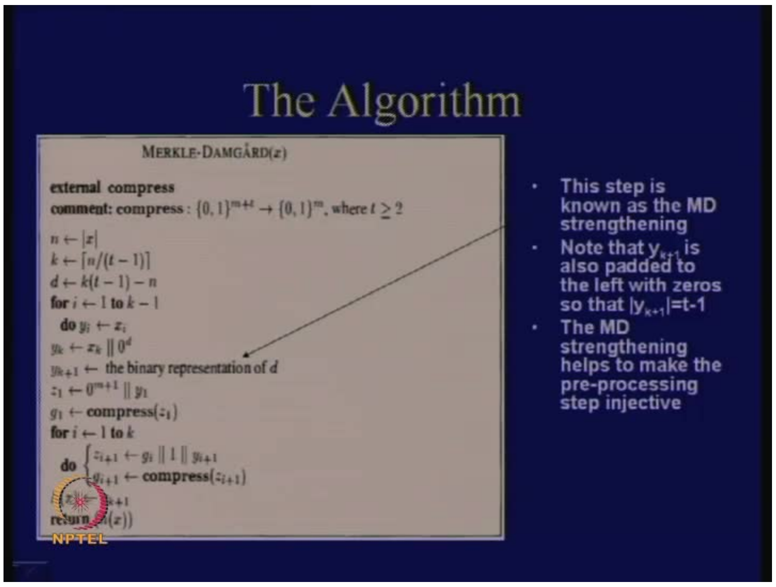
The proof idea is like this, so before I go into the proof, I will discuss about the construction, but I will just give you the idea behind the - I mean what I mean by the term proof. Proof means that - your problem is now to create a hash function, which is collision resistant. Now, this is a quite big problem, because what you are saying is that you will take a very large string, you will compress that into a small m bit string, which has to be still collision resistant.

The beauty of this Markle Damgard construction is that it reduces this problem to the collision resistance of the compress function. The compress function is much smaller problem, it is just an m plus t bit input, which you are compressing into an m bit output. It gives you a proof that if you are able to make a compress function, which is collision resistant, then your hash function is also collision resistant. So that is the basic elegance of the Markle Damgard construction.

Now, let us go through the steps of the construction before going into the proof. You see that x is again taken as the k bit value like x 1, x 2 and so on. All of them are in this case - I mean, all the size of each particular block is actually equal to t minus 1, so it is not t as we saw in general overview of the construction. It is actually equal to t minus 1, in the last block, is actually of size of t minus 1 minus d.

In order to make it t minus 1 size, what do I have to do? I have to add d 0s. Therefore, in this case, the size of d is actually lesser than equal to t minus 2 that is the way how you are breaking it. What is the value of k? So, k will be in this case, you already have your size as n, but if you have padded that with d blocks, it becomes n plus d. Therefore, n plus d divided by t minus 1 should be the value of k and that you can represent as ceil of n divided by t minus 1.

(Refer Slide Time: 23:52)



The Algorithm

```
MERKLE-DAMGÅRD($x$)

external compress
comment: compress : $\{0,1\}^{m+t} \rightarrow \{0,1\}^m$, where $t \geq 2$
$n \leftarrow |x|$
$k \leftarrow \lceil n/(t-1) \rceil$
$d \leftarrow k(t-1) - n$
for $i \leftarrow 1$ to $k-1$
  do $y_i \leftarrow x_i$
$y_k \leftarrow x_k \,\|\, 0^d$
$y_{k+1} \leftarrow$ the binary representation of $d$
$z_1 \leftarrow 0^{m+1} \,\|\, y_1$
$g_1 \leftarrow$ compress($z_1$)
for $i \leftarrow 1$ to $k$
  do $\begin{cases} z_{i+1} \leftarrow g_i \,\|\, 1 \,\|\, y_{i+1} \\ g_{i+1} \leftarrow \text{compress}(z_{i+1}) \end{cases}$
$h(x) \leftarrow g_{k+1}$
return $(h(x))$
```

- This step is known as the MD strengthening
- Note that y_{k+1} is also padded to the left with zeros so that $|y_{k+1}|$=t-1
- The MD strengthening helps to make the pre-processing step injective

Now, let us see the construction. The construction you have to go through this algorithm little bit. You see that 0 1 m plus t bit is to 0 1 m bit is my constructions, complex function which we have taken. Here, we are assumed that t is actually greater than equal to 2, but ideally t should be actually greater than equal to 1, but we will be excluding the case t equal to 1 for now. You just consider the case when t is greater than equal to 2, why? Because, if t equal to 1, then this k becomes undefined, therefore we will consider that t is greater than or equal to 2 for this case.

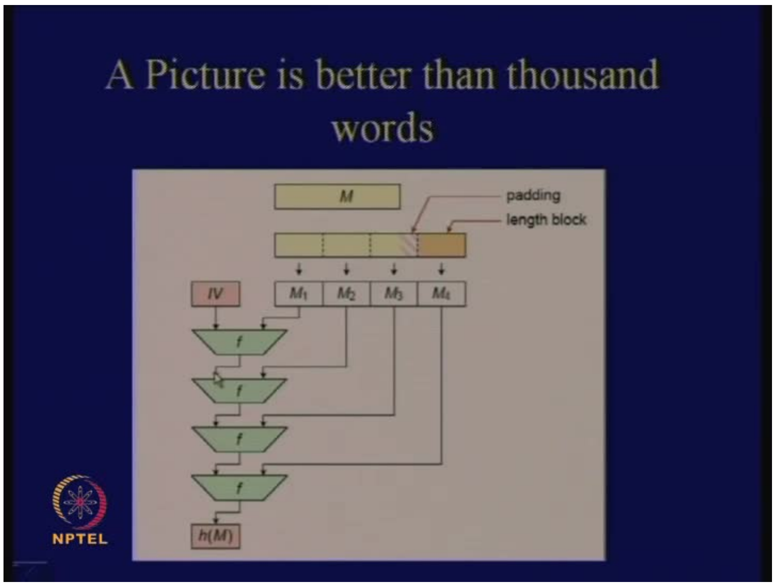
You see that n is actually equal to the size of x and k is equal to n divided by t minus 1 and ceil of that. What you do is that for each of these steps, you keep on doing certain operations. What you do is like this, therefore you see that for i equal to 1 to k minus 1, you take x i, assign that to y i, so that is the preprocessing step. Only the kth block what you do is that you take x k to the right, you pad that with d 0s. If you do that then this also becomes of size t minus 1, then there is an additional step, which is actually called Markle Damgard's strengthening step; therefore, this is called m d strengthening.

What is that m d strengthening step? What you do is this that is you take y k plus 1; in y k plus 1, you write the binary representation of d. So, d is some value some 4, 5, 6 something. You write the binary representation of d, you pad to the left with 0s, because you have to make this also of size t minus 1. So, all the y values are of size t minus 1, so this is the way how the y's.

Then, what you do is that you start your operation. What you do? You take the first one, so how many bits are there in y 1, t minus 1? You pad that with m plus 1 0s, therefore size becomes equal to m plus t, because there are m plus 1 values here and there are t minus 1 values here, so there are m plus t values here. So that means you can feed it to the compress function, if you feed it to the compress function, it will give you back an m bit value, call that g 1.

Now, you continue the iterative process, which means that you take g i, which you have obtained in the previous step. You know, you have also obtained y i plus 1, in between you keep a 1. Therefore, what is the size of this particular string? m plus t, again you can feed this into the complex function and obtain the m bit output. This you can iterate, you can obtain the corresponding value, values like g 1, g 2, g 3 and so on till g k plus 1. Therefore, this final value of g k plus 1 is actually the hash value that is the h x value.

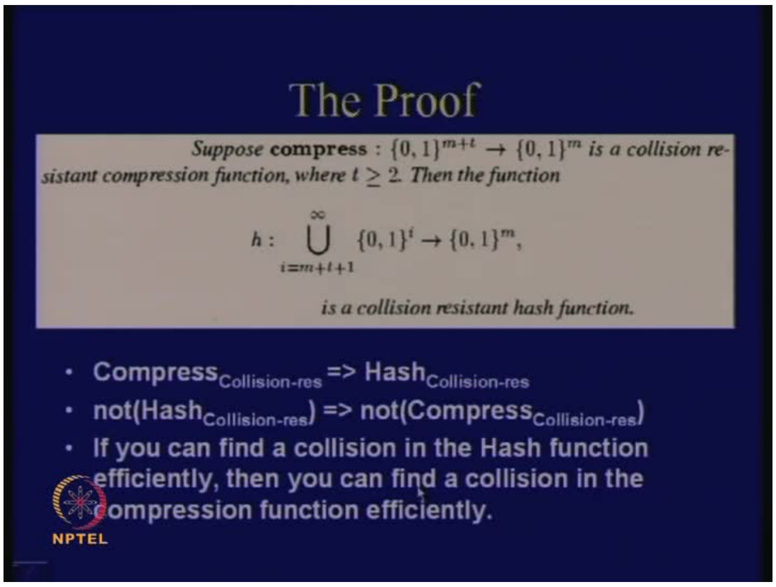
(Refer Slide Time: 27:21)


A Picture is better than thousand words

You note that this is I mean sort of a pseudo algorithm way of representing the story, but diagrammatically this will look like this. Therefore, what you do is that you take this value of m, then do an additional padding to the kth block, then finally you add the length block also, which is the y k plus 1th block in the pseudo algorithm. Then you are taking each m 1, m 2, m 3 and so on, this is the IV value in my pseudo algorithm, I have taken this to be 0 m plus 1, but that is the public value anyway.

Then, what you do is that if you feed it to the complex function, you obtain an m bit output, you take the next t minus 1, but in between you have to add 1 here also and you continue with that to obtain the corresponding hash value. So, all the hash functions like sha, m d 5, m d 4, this follows these principles.

(Refer Slide Time: 28:12)



The Proof

Suppose **compress** : $\{0,1\}^{m+t} \rightarrow \{0,1\}^m$ *is a collision resistant compression function, where* $t \geq 2$. *Then the function*

$$h : \bigcup_{i=m+t+1}^{\infty} \{0,1\}^i \rightarrow \{0,1\}^m,$$

is a collision resistant hash function.

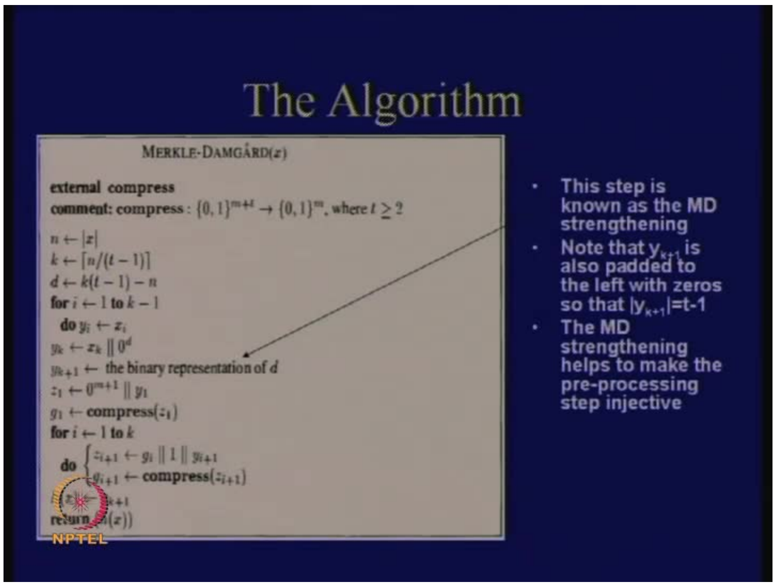
- $\text{Compress}_{\text{Collision-res}} \Rightarrow \text{Hash}_{\text{Collision-res}}$
- $\text{not}(\text{Hash}_{\text{Collision-res}}) \Rightarrow \text{not}(\text{Compress}_{\text{Collision-res}})$
- If you can find a collision in the Hash function efficiently, then you can find a collision in the compression function efficiently.

Now, we come to the slightly little bit more complicated part, but it is quite simple proof, which says that suppose compress 0 1 m plus t to 0 1 m, is a collision resistant compress function, then the function, this is actually a collision resistant hash function. This is the representation of the collision function, you see that your 0 1 string can actually lie from m plus t plus 1 to infinity. The domain can be 0 1 - I mean from m plus t plus 1 to infinity and that you are compressing to an m bit output.

What you have to prove is that if these compresses function is a collision resistant function, then I mean this are a collision resistant function, then your hash function is also collision resistant. Like the previous day's class, what you have assumed is that you will take a not in both sides, therefore if we assume that the hash function is not collision resistance that means, if you have assumed that there is a collision which we have obtained for the hash function, then we can prove that the compress function also has a hash function, I mean also as a collision

If you can find a collision in the hash function efficiently, so efficiently is again under my definition of efficiency. Then, you can find a collision in the compression function also efficiently. How do we prove this?

(Refer Slide Time: 29:44)



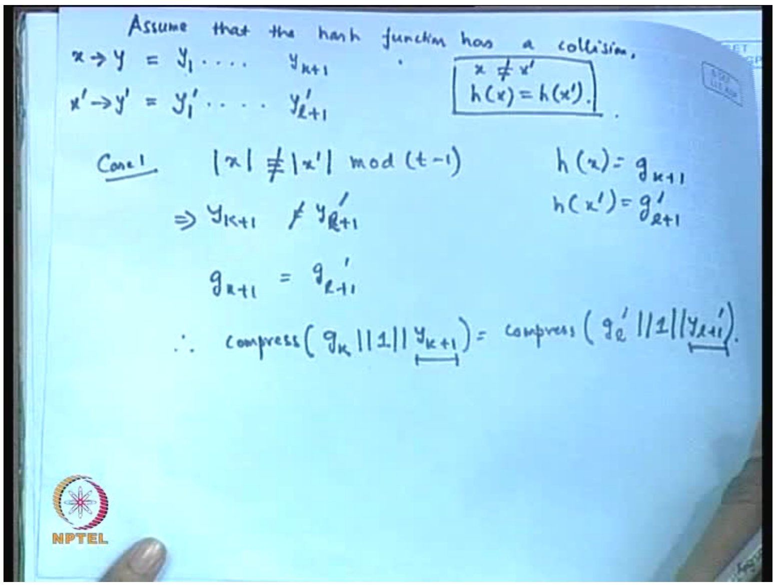
The Algorithm

```
MERKLE-DAMGÅRD(x)
external compress
comment: compress : $\{0,1\}^{m+t} \to \{0,1\}^m$, where $t \geq 2$
$n \leftarrow |x|$
$k \leftarrow \lceil n/(t-1) \rceil$
$d \leftarrow k(t-1) - n$
for $i \leftarrow 1$ to $k-1$
  do $y_i \leftarrow x_i$
$y_k \leftarrow x_k \| 0^d$
$y_{k+1} \leftarrow$ the binary representation of $d$
$z_1 \leftarrow 0^{m+1} \| y_1$
$g_1 \leftarrow$ compress($z_1$)
for $i \leftarrow 1$ to $k$
  do { $z_{i+1} \leftarrow g_i \| 1 \| y_{i+1}$
       $g_{i+1} \leftarrow$ compress($z_{i+1}$)
[illegible] $_{k+1}$
return [illegible]$(x)$)
```

- This step is known as the MD strengthening
- Note that y_{k+1} is also padded to the left with zeros so that $|y_{k+1}|=t-1$
- The MD strengthening helps to make the pre-processing step injective

In order to prove this, let us go back to my algorithm and the concentrate upon certain cases.

(Refer Slide Time: 29:54)



Assume that the hash function has a collision.

$x \to y = y_1 \ldots \; y_{k+1}$

$x' \to y' = y_1' \ldots \; y_{l+1}'$

$$\boxed{\begin{array}{c} x \neq x' \\ h(x) = h(x'). \end{array}}$$

Case 1 $\quad |x| \not\equiv |x'| \bmod (t-1)$ $\qquad h(x) = g_{k+1}$

$\Rightarrow y_{k+1} \neq y_{l+1}'$ $\qquad h(x') = g_{l+1}'$

$g_{k+1} = g_{l+1}'$

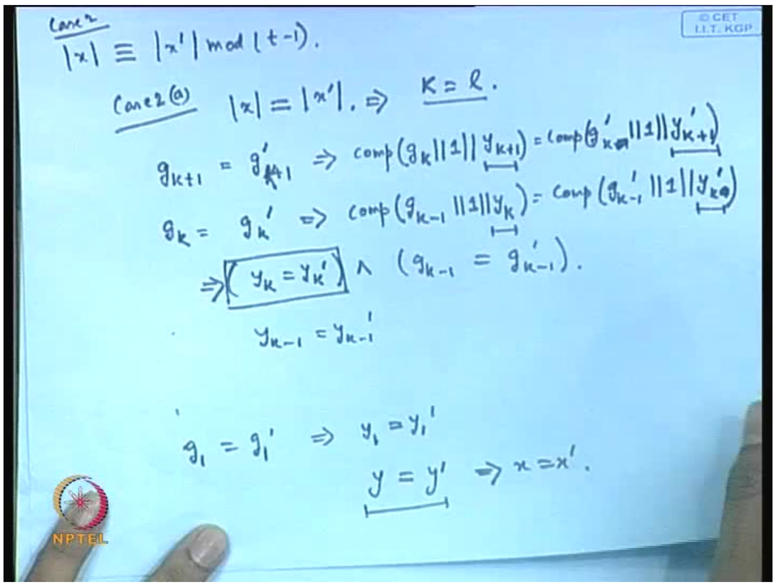
$\therefore \text{compress}(g_k \| 1 \| \underbrace{y_{k+1}}) = \text{compress}(g_l' \| 1 \| \underbrace{y_{l+1}'})$.

You see that both my algorithm are essentially, I have got y 1, say y k plus 1. Now, assume that we have been able to find out a collision for the hash function. What does it mean? It means that you have found out two values, if you assume that the hash functions has a collision, so that means that x and x dash which are not the same, has got h x dash as seen. You have find out two such values which are like this.

That means, now if you concentrate upon say y 1, I mean concentrate suppose on x and x dash, concentrate upon I mean the two preprocessing steps of x and x dash. Therefore, assume that x results in y and x dash results in y dash. What is y? The y is suppose y 1 to y k plus 1, y dash is y 1 dash to y l plus 1 dash. This can be two arbitrary lens, based upon which we have done this operation.

Now, assume two cases: case 1 is when mod of x and mod of x dash mod of t minus 1 and not the same, which means, if you take mod x and mod x dash mod of t minus 1 and not the same, you know, it means that if I take a mod of t minus 1, so that is the final number of values, which you are padding. So that is minus being both the cases, minus d 1, minus d and minus d dash value. In this case, what I am saying is the d and d dash is not the same.

(Refer Slide Time: 33:57)



Case 2

$|x| \equiv |x'| \bmod (t-1).$

Case 2(a) $|x| = |x'|. \Rightarrow \underline{K = l}.$

$g_{k+1} = g'_{k+1} \Rightarrow comp(g_k \| 1 \| y_{k+1}) = comp(g'_{k} \| 1 \| y'_{k+1})$

$g_k = g'_k \Rightarrow comp(g_{k-1} \| 1 \| y_k) = comp(g'_{k-1} \| 1 \| y'_k)$

$\Rightarrow \boxed{(y_k = y'_k)} \wedge (g_{k-1} = g'_{k-1}).$

$y_{k-1} = y'_{k-1}$

$g_1 = g'_1 \Rightarrow y_1 = y'_1$

$\underline{y = y'} \Rightarrow x = x'.$

© CET I.I.T. KGP

Therefore, y k plus 1 and y k plus 1 dash or other y l plus 1 dash are different. Now, consider upon the hash output. So, what was my hash stream, which was outputted - I mean what was my corresponding hash value? The h x was equal to g k plus 1 and h x dash was equal to g of l plus 1, so call that g dash of l plus 1. If h x and h x dash are the same values, then g of k plus 1 will be equal to g of l plus 1 dash.

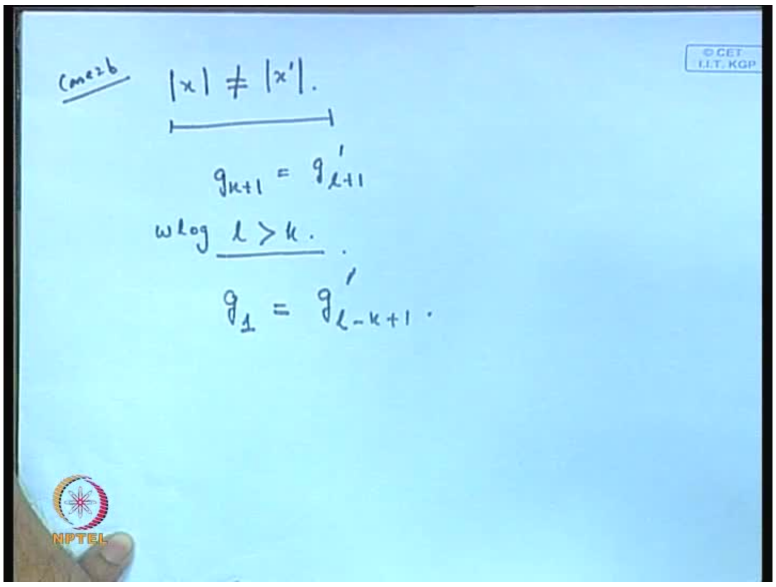
Therefore, what does it mean? It means that compresses g of k with 1 with y k plus 1, will be equal to compress g l dash 1 y l plus 1 dash. Now we know y k plus 1 and y l plus 1 dash are different. What does it mean? It means that we are been able to find out two values which are giving you the same compress outputs. Therefore, what you are showing is that the compression function is not a collision resistance, this solves one case.

What about the other case now? Is this part clear? The other case is when mod of x and mod of x dash mod of t minus 1 are same. This actually will be easy if I break this into two parts. So, I call that first part as mod of x - the first case is mod of x is equal to mod of x dash, which means that what does it imply? It implies that k and l are same values.

Now, if you observe that what your saying is g of k plus 1 is equal to g dash l plus 1 or other, l will be in this case k itself, so g dash of k plus 1. So, what does it mean? It means that compress I call that comp for private, so comp will be g of k 1, here we have got y of k plus 1 is equal to comp g of k 1 and y of k plus 1 dash.

Here, you note that y of k plus 1 and y of k plus 1 dash are the same values - I mean, if you say that this is not a collision for the compress function, definitely g of k and g of k dash should be equal. Therefore, g of k and g of k dash must be equal. So that be the case and again compress, we can continue like this. This becomes g of k minus 1, 1 y of k and that is equal of compress g of k minus 1 dash, then you have got a 1 here, y k plus y k dash; so this is the idea.

(Refer Slide Time: 38:09)


Case 2b $|x| \neq |x'|$.

$g_{k+1} = g'_{l+1}$

wlog $l > k$.

$g_1 = g'_{l-k+1}$.

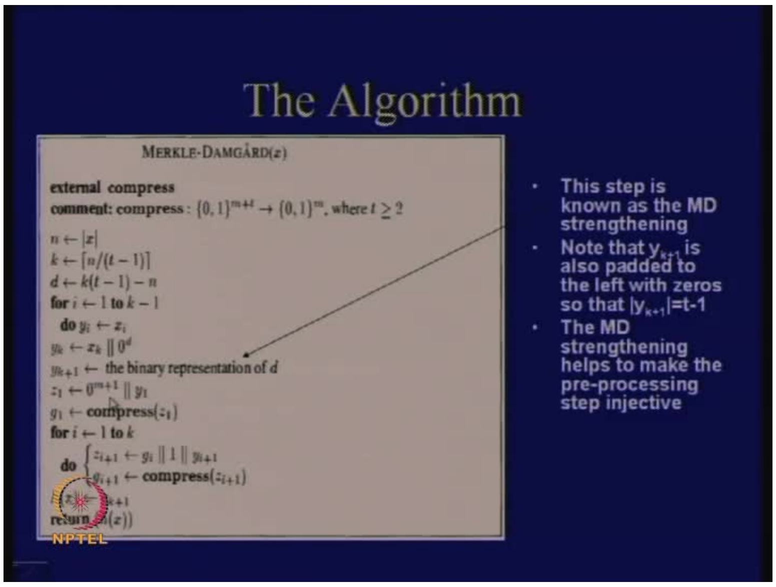
Now, you see that what you have? If you have, still do not want to force a collision of the compress function, then two things become clear: one is that y of k and y of k dash must be same. Therefore, if this is not a collision, then this implies that y of k is equal to y k dash and g of k minus 1 is equal to g of k minus 1 dash.

Now, if I assume that g of k minus 1 and g of k dash minus 1 are not the same, then again I can continue in this fashion. If I continue, since my lens at both are the same k and l are same, I will continue till g 1 equal to g 1 dash. In each case, other will result like this, like y k equal to y k dash and again, y k minus 1 equal to y k minus 1 dash and finally, we have y 1 equal to y 1 dash.

What we are saying now is that y and y dash are the same, because for each sub part y is having been same in both the cases. If y and y dash are the same, because my preprocessing step was my injective transformation, so this implies that x and x dash are the same values, but that contradicts my assumption, because I will assume that x and x dash are different, therefore you see a contradiction, you follow? This part is also solved, therefore there must be a collision in the compress function and you are able to show this reduction.

What about the other case which we have? The other part case 2b, is mod of x, is not equal to mod of x dash. This proof is actually quite similar to the case 2a, expect for the fact that you have got g of k plus 1 equal to g of l plus 1 dash. If I assume without loss of generality that l is greater than k and then I can continue in this fashion, what will obtain is g of 1 will be equal to g of l minus k plus 1 dash. What is g 1, what is g of l minus k plus 1 dash? If you see you are algorithm here, just see your algorithm, how it starts.

(Refer Slide Time: 39:03)

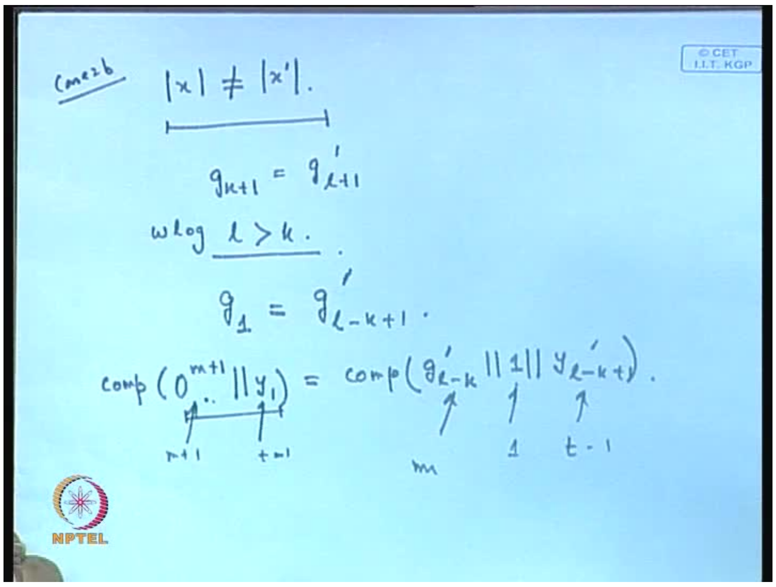


The Algorithm

```
MERKLE-DAMGÅRD(x)

external compress
comment: compress : $\{0,1\}^{m+t} \to \{0,1\}^m$, where $t \geq 2$
$n \leftarrow |x|$
$k \leftarrow \lceil n/(t-1) \rceil$
$d \leftarrow k(t-1) - n$
for $i \leftarrow 1$ to $k-1$
    do $y_i \leftarrow x_i$
$y_k \leftarrow x_k \| 0^d$
$y_{k+1} \leftarrow$ the binary representation of $d$
$z_1 \leftarrow 0^{m+1} \| y_1$
$g_1 \leftarrow$ compress$(z_1)$
for $i \leftarrow 1$ to $k$
    do { $z_{i+1} \leftarrow g_i \| 1 \| y_{i+1}$
         $g_{i+1} \leftarrow$ compress$(z_{i+1})$
[illegible]$_{k+1}$
return [illegible]$(x))$
```

- This step is known as the MD strengthening
- Note that y_{k+1} is also padded to the left with zeros so that $|y_{k+1}|$=t-1
- The MD strengthening helps to make the pre-processing step injective

(Refer Slide Time: 39:20)



Case 2b $\quad |x| \neq |x'|$.

$$g_{k+1} = g'_{l+1}$$

wlog $l > k$.

$$g_1 = g'_{l-k+1}.$$

$$\text{comp}(\underset{m+1}{0^{m+1}} \| \underset{t-1}{y_1}) = \text{comp}(\underset{m}{g'_{l-k}} \| \underset{1}{1} \| \underset{t-1}{y'_{l-k+1}}).$$

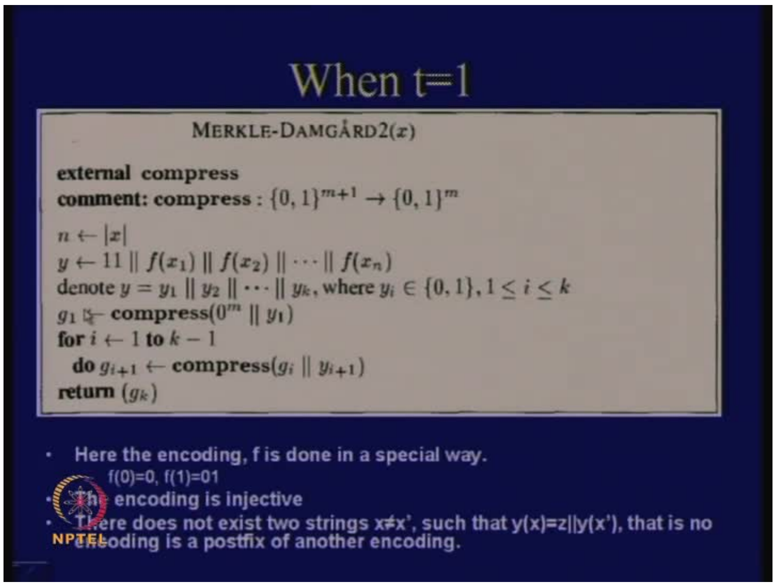
The first step is that you take 0 m plus 1 and you concatenate between y 1, so that is what you are compressing to obtain g 1. What you are obtaining is 0 of m plus 1 concatenated with y 1 and that is what you are compressing. What about this? This is simple like - you take g of l minus k, concatenate with y 1 and you have got y of l minus k plus 1 dash.

Now, if you compare these two things, you observe the m plus first bit, the m plus first bit in this case is what? y 1 is how many bits? It is of t minus t minus 1 bits; this is of m plus 1 bits. How many bits are there in g dash l minus k? m bits. So, there is 1 bit here; how many bits are there? t minus 1.

Now, observe the m plus first bit; m plus first bit here is 0, but what about the m plus first bit here, it is 1, therefore they are definitely different. Therefore, you have essentially again found out a collision in the compress function. Now, you see why 1 was used and why 0 m plus 1 was used here, that you understand from the proof actually. This gives you an idea that actually your collision resistance of your compressed function, implies there your iterated hash function is also collision resistant.

This is again under the ideal assumptions, so we are again assuming that each hash function is an independent computation. If you find that there are dependencies in the hash functions, then all this proof systems will come down.

(Refer Slide Time: 41:12)



When t=1

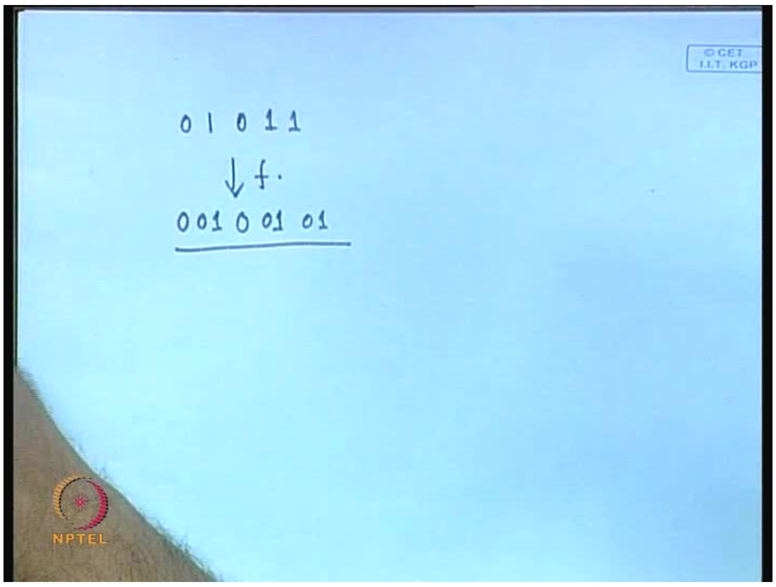
```
MERKLE-DAMGÅRD2($x$)

external compress
comment: compress : $\{0,1\}^{m+1} \rightarrow \{0,1\}^m$
$n \leftarrow |x|$
$y \leftarrow 11 \,||\, f(x_1) \,||\, f(x_2) \,||\, \cdots \,||\, f(x_n)$
denote $y = y_1 \,||\, y_2 \,||\, \cdots \,||\, y_k$, where $y_i \in \{0,1\}, 1 \leq i \leq k$
$g_1 \leftarrow$ compress($0^m \,||\, y_1$)
for $i \leftarrow 1$ to $k-1$
  do $g_{i+1} \leftarrow$ compress($g_i \,||\, y_{i+1}$)
return ($g_k$)
```

- Here the encoding, f is done in a special way.
  f(0)=0, f(1)=01
- The encoding is injective
- There does not exist two strings x≠x', such that y(x)=z||y(x'), that is no encoding is a postfix of another encoding.

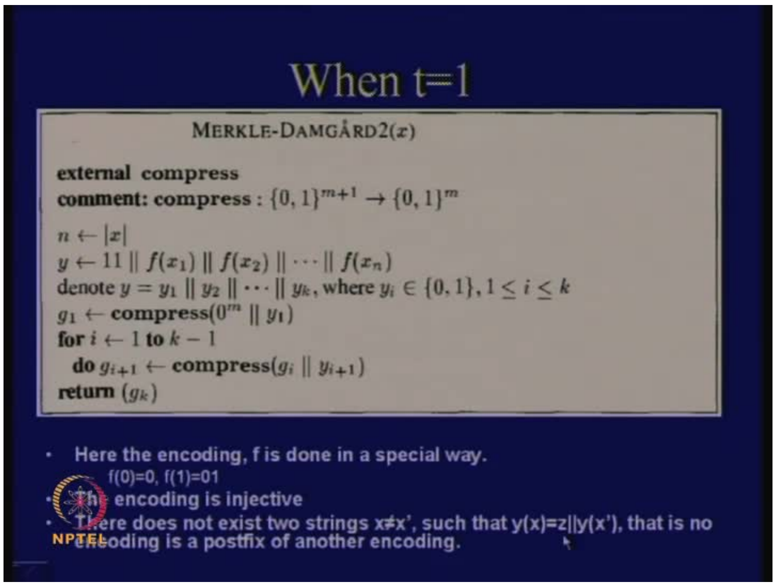
Now, we will just conclude with the case when t equal to 1. So, I think this part is clear to us, let us consider the case when t equal to 1. When t is equal to 1 that means that you will take a compress function 0 1 m plus 1 to 0 1 m, this is how you do. You take x, the preprocessing step is slightly different. So, what you do is that you take 1 1 and after that you take any function f - I mean not any function, a particular function and then apply f x 1, f x 2 and so on. You see that mod x means - n means what? How many bits are there?

The n bits are there. For each of the bit you are computing f x 1, f x 2, and so on. This is how you do the encoding, if the input is 0, then the output is also 0, but if the input is 1, then the output is 0 1.

(Refer Slide Time: 42:18)



(Refer Slide Time: 42:46)



When t=1

```
MERKLE-DAMGÅRD2($x$)

external compress
comment: compress : $\{0,1\}^{m+1} \rightarrow \{0,1\}^m$
$n \leftarrow |x|$
$y \leftarrow 11 \| f(x_1) \| f(x_2) \| \cdots \| f(x_n)$
denote $y = y_1 \| y_2 \| \cdots \| y_k$, where $y_i \in \{0,1\}, 1 \leq i \leq k$
$g_1 \leftarrow$ compress($0^m \| y_1$)
for $i \leftarrow 1$ to $k-1$
  do $g_{i+1} \leftarrow$ compress($g_i \| y_{i+1}$)
return ($g_k$)
```

- Here the encoding, f is done in a special way.
  f(0)=0, f(1)=01
- The encoding is injective
- There does not exist two strings x≠x', such that y(x)=z||y(x'), that is no encoding is a postfix of another encoding.

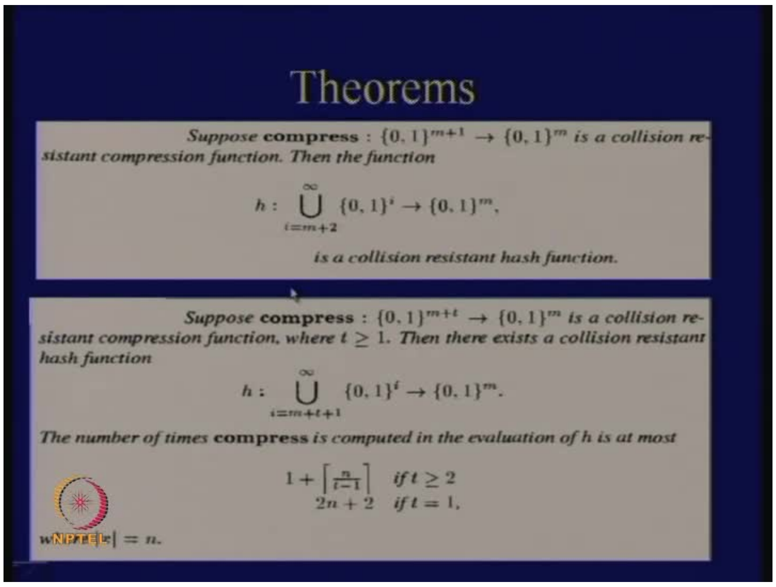
Do you understand the encoding? Suppose, your input is a string like 0 1 0 1 1 like this, then if you are doing this hash encoding f, then for your 0 you get 0, for your 1 you get 0 1, for your 0 again get 0, for 1 you get 0 1, for again 1 you get 0 1; so this is the way how you are encoding the input x. Now, you see that in this case, you take this value of x and you start encoding this, the only the thing is that the first two bits are 1.

So, why do you keep these two things? Two things are important again, the encoding is an injective encoding. The other thing is that it there does not exist two strings: x and x dash which are different, such that y of x and y of x dash satisfy this kind of relation. That is y of x is equal to z, which is any arbitrary string, concatenated with y of x dash. You see that this can never happen, that is no encoding is a postfix of another encoding, why? Then, because y of x, will obviously start with two ones, y of x dash will also start with two ones, but in this y x, two ones can never occur in between; two successive ones can never occur in between. Therefore, this relation will never be satisfied, do you follow?

What I am saying is that if you take two values of x and x dash, there can never be the case where y of x dash and y of x dash satisfies this relation; that is y of x dash is a postfix of another encoding, because of the reason that two consecutive ones can never occur in the preprocess step. It is quite simple, therefore you obtain this y 1 and then, each of them is of now of 1 bit.

So, what you do is that you take 0 m, I mean m bits and this is of - y 1 is of 1 bit, so that is m plus 1 bits, you compress, you obtain g 1, you continue this process. You take g I, concatenate with y i plus 1, so g i is m bit and y i plus 1 is 1 bit, so this is m plus 1 bit, you engage this compress function, which gives you an m bit output form and m plus 1 bit output, obtain m bit output. Finally, you are returning g k as the corresponding hash value.

(Refer Slide Time: 45:03)



Theorems

Suppose **compress** : $\{0,1\}^{m+1} \to \{0,1\}^m$ *is a collision resistant compression function. Then the function*

$$h : \bigcup_{i=m+2}^{\infty} \{0,1\}^i \to \{0,1\}^m,$$

is a collision resistant hash function.

Suppose **compress** : $\{0,1\}^{m+t} \to \{0,1\}^m$ *is a collision resistant compression function, where* $t \geq 1$. *Then there exists a collision resistant hash function*

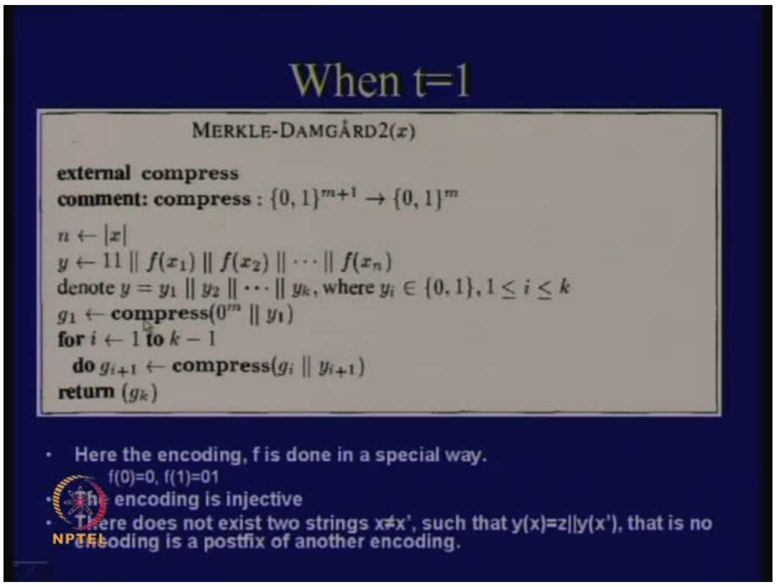
$$h : \bigcup_{i=m+t+1}^{\infty} \{0,1\}^i \to \{0,1\}^m.$$

The number of times **compress** *is computed in the evaluation of h is at most*

$$\begin{array}{ll} 1 + \left\lceil \frac{n}{t-1} \right\rceil & \text{if } t \geq 2 \\ 2n + 2 & \text{if } t = 1, \end{array}$$

where $|x| = n$.

(Refer Slide Time: 45:12)



When t=1

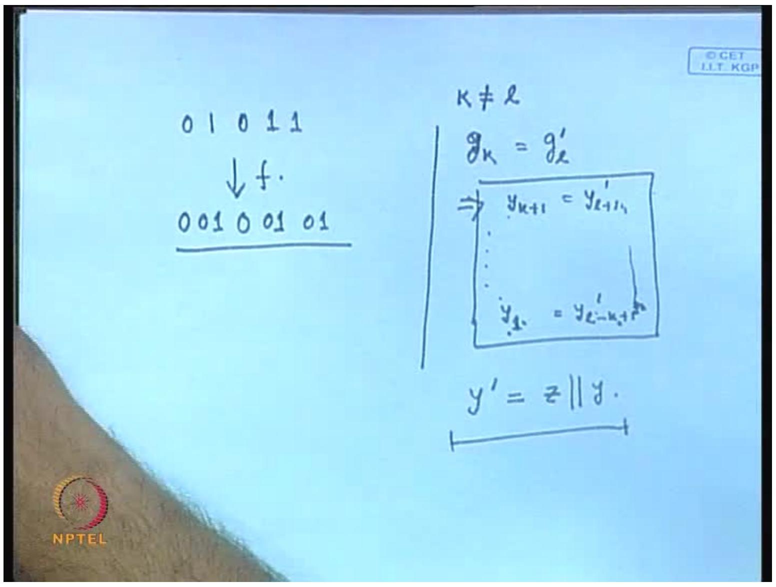
```
MERKLE-DAMGÅRD2($x$)
external compress
comment: compress : $\{0,1\}^{m+1} \to \{0,1\}^m$
$n \leftarrow |x|$
$y \leftarrow 11 \| f(x_1) \| f(x_2) \| \cdots \| f(x_n)$
denote $y = y_1 \| y_2 \| \cdots \| y_k$, where $y_i \in \{0,1\}, 1 \leq i \leq k$
$g_1 \leftarrow$ compress($0^m \| y_1$)
for $i \leftarrow 1$ to $k - 1$
  do $g_{i+1} \leftarrow$ compress($g_i \| y_{i+1}$)
return ($g_k$)
```

- Here the encoding, f is done in a special way.
  - f(0)=0, f(1)=01
- The encoding is injective
- There does not exist two strings $x \neq x'$, such that $y(x)=z||y(x')$, that is no encoding is a postfix of another encoding.

Now, this we can actually show to be collision resistant, the proof is quite very simple, I will not go into the details of the proof, you can find out the details from Hinson's book. The reason is that - I mean, if you again you divide into two cases. Again, you assume that there is a collision for the hash function, which means that there are two x values: x and x dash, which are distinct and which results in the same hashed outputs.

Now, again assume that x and x dash, the sizes are same or the sizes are different. If the sizes are different, then the proof is exactly - I mean if the sizes are same, then the proof is exactly same. If the sizes are same, then again you can assume that g k equal to g r l dash. Similarly, you can again continue and show that y and y dash will be same, you can continue in that fashion, but what about the other case when the sizes are not the same?

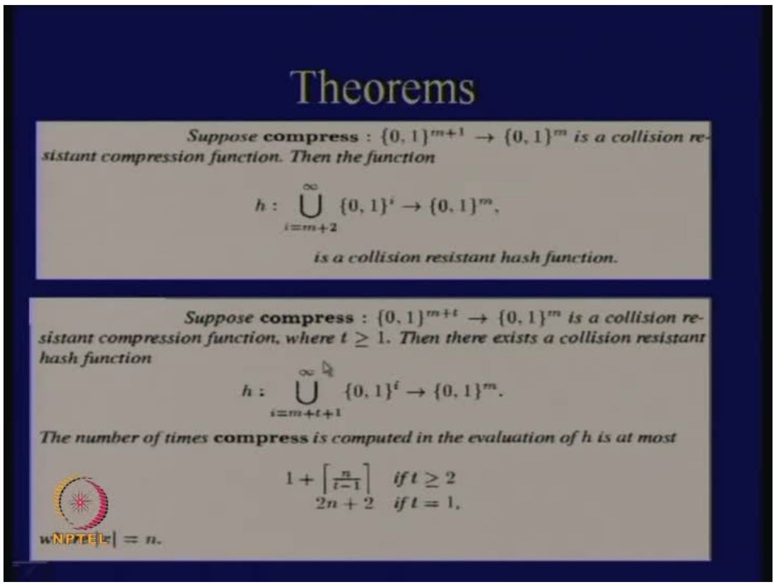
(Refer Slide Time: 46:05)



If the sizes are not the same, then you will actually show that - if the sizes are not the same, which means, if I assume that the size in one case is k and the size is another case is l, they are not the same. What you do is that you continue from y k, you start with g k and g l dash. So, if g k and g l dash are the same, because you are assuming that the hash function collides with for two x values: x and x dash, then g k equal to g l dash, means what? The g k equal to g l dash means that correspondingly y k plus 1 will be equal to y of l plus 1 dash.

Similarly, you can continue, finally you will have y 1 equal to y of l minus k plus 1 dash. Now, this means that you are violating the postfix assumption, because here, you have got one string, which is actually the postfix of the other string. You see that this is a shorter string, this is a bigger string for y's, do you see that? This is a y string in one case, this is the y string in another case, this actually forms a post part of - I mean this is actually a bigger string. Therefore, if you considered y dash, you can write y dash as some z, which is followed by a y string. Therefore, this is actually violating my property of the encoding function. This cannot happen, therefore again you are proving a contradiction, I mean, showing a contradiction. This establishes the fact that if the compress function is collision resistant, then your hash function is also a collision resistance.

(Refer Slide Time: 47:58)



Theorems

Suppose **compress** : $\{0,1\}^{m+1} \rightarrow \{0,1\}^m$ *is a collision resistant compression function. Then the function*

$$h : \bigcup_{i=m+2}^{\infty} \{0,1\}^i \rightarrow \{0,1\}^m,$$

is a collision resistant hash function.

Suppose **compress** : $\{0,1\}^{m+t} \rightarrow \{0,1\}^m$ *is a collision resistant compression function, where* $t \geq 1$. *Then there exists a collision resistant hash function*

$$h : \bigcup_{i=m+t+1}^{\infty} \{0,1\}^i \rightarrow \{0,1\}^m.$$

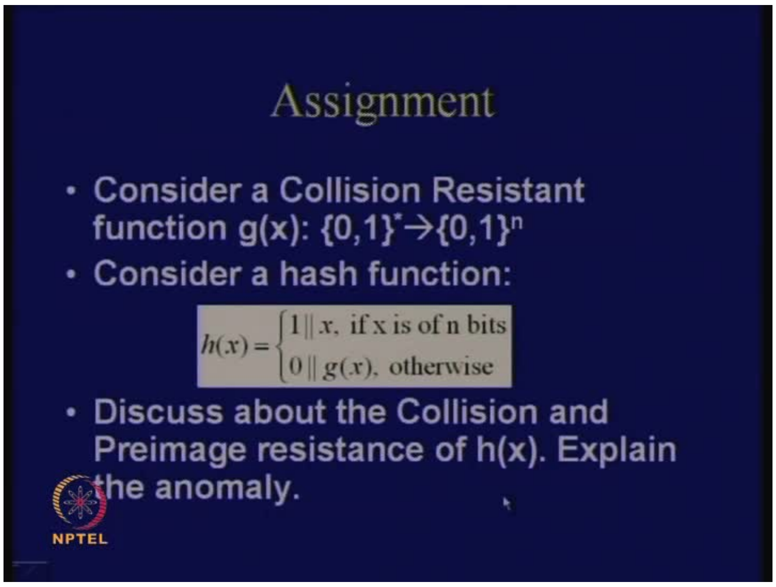
The number of times **compress** *is computed in the evaluation of h is at most*

$$\begin{array}{ll} 1 + \left\lceil \frac{n}{t-1} \right\rceil & \text{if } t \geq 2 \\ 2n + 2 & \text{if } t = 1, \end{array}$$

w[illegible]$| = n$.

This is what the theorem says, if you put the two theorems together, then now you have actually got t greater than equal to one, because the previous case was t greater equal to two and this case is equal to t equal to one. So, this is what you have shown that if the compressed function is a collision resistant compress function, then so is the hash function. You can actually compute the number of times the compress is computed in each of the cases, this works out as follows. These are quite simple calculations, which we can just check how many times the compressed functions are being engaged.

(Refer Slide Time: 48:35)



Assignment

- Consider a Collision Resistant function g(x): $\{0,1\}^* \rightarrow \{0,1\}^n$
- Consider a hash function:

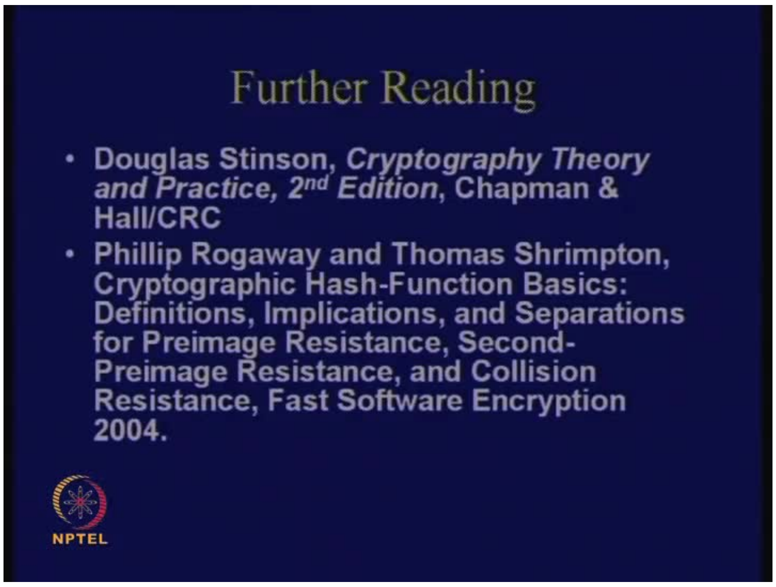
$$h(x) = \begin{cases} 1 \| x, & \text{if x is of n bits} \\ 0 \| g(x), & \text{otherwise} \end{cases}$$

- Discuss about the Collision and Preimage resistance of h(x). Explain the anomaly.

This is an assignment for you again, therefore you see that consider a collision resistant function g x, which takes an infinite string and results in a 0 1 n bit output. Consider a hash function h x, which is one concatenate with x. If x is of n bits, otherwise it is 0 concatenate with g x. Now, the question is that you have to discuss about the collision and the preimage resistance of h x. You see that since g x is collision resistant, h x is also collision resistance, but what about the preimage problem? You see that in one case, you have found out that there is 1 as an output. Therefore, if I assume that is 50 percent of time, you will get 1 and 50 percent of time, you will get 0. If you get 1 concatenate with x and then the inverse is through so clear, with the probability of at least 50 percent, you are able to invert the function h x.

So, what does it mean? This is not preimage resistant, but what have we proved? We are proved that collision resistance implies preimage resistance. So, there is the anomaly, various small problems in this particular example from what we have proved in the class, so you are supposed to justify that.

(Refer Slide Time: 50:01)



Further Reading

- Douglas Stinson, *Cryptography Theory and Practice, 2nd Edition*, Chapman & Hall/CRC
- Phillip Rogaway and Thomas Shrimpton, Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance, Fast Software Encryption 2004.

Next time, we will again continue with the hash function, this is the textbook that we have followed and this is some paper that I have followed. This is the main book, you can refer to this textbook and you will find these details here. So, we will again continue with cryptographic hash functions next time.