

Foundations of Cryptography
Prof. Dr. Ashish Choudhury
(Former) Infosys Foundation Career Development Chair Professor
Indian Institute of Technology – Bangalore

Lecture – 30
Generic Attacks on Hash Functions and Additional Applications

Hello everyone, welcome to this lecture. Just a quick recap. In the last lecture, we had seen how to use collision-resistant hash function to create message authentication codes for arbitrary long messages.

(Refer Slide Time: 00:31)

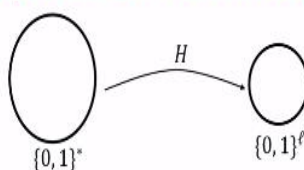
Roadmap

- ❑ Birthday attacks for finding collisions in hash functions
- ❑ Small space Birthday attacks for finding collisions in hash functions
- ❑ Applications of hash functions

So, the plan for this lecture is as follows. In this lecture, we will discuss birthday attacks, which is a class of generic attacks which can be launched against any hash functions and we will also discuss the small space birthday attacks for finding collisions in hash functions, and finally, we will discuss some of the other applications of hash functions apart from the message authentication codes.

(Refer Slide Time: 01:03)

Birthday Attacks for Finding Collisions



Naive collision finding algorithm for H: <ul style="list-style-type: none"> ❖ Evaluate H at $2^l + 1$ distinct inputs ➤ success probability 1 ➤ Running time $\mathcal{O}(2^l)$ 	Generalization of the naive algorithm: <ul style="list-style-type: none"> ❖ Evaluate H at q distinct inputs x_1, \dots, x_q ❖ Output x_i, x_j if $H(x_i) = H(x_j)$ ❖ Success probability of finding a collision?
Success probability of the generalized algorithm computed <u>assuming H as a random function</u> <ul style="list-style-type: none"> ❖ Worst possible case for finding a collision ❖ Probability of a collision gets better for a non-random H and a random set of queries 	

So let us start with birthday attacks for finding collisions. So, imagine you are given a deterministic hash function which is publicly known. Then the naive algorithm for finding collision in this hash function is as follows: you can evaluate this hash function at 2^{l+1} number of distinct inputs and it is guaranteed that with 100% success you will obtain a collision. The success probability of finding the collision here is 1. This is because of the pigeonhole principle because at the best you can hope that 2^l distinct inputs go to 2^l distinct outputs, but since you are evaluating a hash function at more input than 2^l inputs, then definitely there will be 2 inputs which will be giving you the same hash value and that will give you collision.

So, the success probability of this naïve collision finding algorithm is 1, it gives you 100% success guarantee, but unfortunately the running time of this naive algorithm is of order to 2^l and if l is significantly large, then of course this naïve collision finding algorithm is impractical.

So now let us see a generalization of this naive algorithm and the generalization goes as follows: you evaluate this hash function at q distinct inputs, which I denote by x_1, \dots, x_q and you obtain the hash values and you pass these (input, output) values and check whether exists a pair (x_i, x_j) such that your hash values are same. Now, we are interested to argue here what is the success probability of this generalized algorithm?

So, remember if q is said to be 2^{l+1} , then basically this generalized algorithm becomes a naive algorithm in that case, the success probability is 1, but now your q need not be 2^{l+1} . it could be any function and now our goal is to analyze the success probability of this naive

algorithm as a function of q and the size of your hash function. So, while doing the analysis, what we will assume is that we will assume that the function H is behaving like a truly random function, and why so because that is the best you can hope for.

Because if indeed your function H behaves like a truly random function, then that is the most difficult task for the adversary where the adversary's goal there will be to find a collision. Because it can be proved that the probability of finding collision for this generalized algorithm gets better if your underlying function H is a non-random function and if instead of querying on distinct inputs x_1 to x_q , your algorithm makes queries for random set of x values.

So, we are basically not making any stronger assumption by assuming that the function H is a random function because the success probability of this naive algorithm is lower bounded by the success probability that this naive algorithm which will give you the case where we are assuming your underlying function H is a random function.

(Refer Slide Time: 04:30)

Birthday Attacks for Finding Collisions

- $H: \{0, 1\}^* \Rightarrow \mathcal{Y}$ a random function, where $|\mathcal{Y}| = N$
- x_1, \dots, x_q : distinct inputs from $\{0, 1\}^*$ □ y_1, \dots, y_q : random values from \mathcal{Y} , where $y_i = H(x_i)$
- Coll(q, N): event that $y_i = y_j$, where $i \neq j$
- Theorem: Let $q \leq \sqrt{2N}$. Then $\Pr[\text{Coll}(q, N)] \geq \frac{q(q-1)}{2N}$
- ❖ NoColl _{i} : event that $\{y_1, \dots, y_i\}$ are distinct NoColl _{q} = Coll(q, N)
- $\Pr[\text{NoColl}_q] = \Pr[\text{NoColl}_1] \cdot \Pr[\text{NoColl}_2 | \text{NoColl}_1] \cdot \dots \cdot \Pr[\text{NoColl}_q | \text{NoColl}_{q-1}]$
- ❖ $\Pr[\text{NoColl}_1] = 1$ --- as y_1 is distinct
- ❖ $\Pr[\text{NoColl}_{i+1} | \text{NoColl}_i] = 1 - \frac{i}{N} \leq e^{-i/N}$ --- since $1 - x \leq e^{-x}$, for all x
- ❖ $\Pr[\text{NoColl}_q] \leq 1 - \frac{q(q-1)}{4N}$
- ❖ Since $\Pr[\text{Coll}(q, N)] = 1 - \Pr[\text{NoColl}_q]$, we get $\Pr[\text{Coll}(q, N)] \geq \frac{q(q-1)}{4N}$

So, let us set the setting for the generalized case here. So, you are given a truly random function from the set of binary strings to a fixed set domain where the size of the domain is L and say you have made q number of queries for this hash values on distinct inputs x_1 to x_q and the corresponding output values are y_1 to y_q . Since I am assuming my hash function is a random function, each of this y_i values are uniformly random values from the set of N values which my hash function can throw as possible outputs.

I denote the event $\text{Coll}(q, N)$, the event that event that $y_i = y_j$, where $i \neq j$. My goal is to compute a lower bound on the probability of this event $\text{Coll}(q, N)$. So, I want to prove here that if the number of queries that I am making here is upper bounded by $\sqrt{2N}$, then $\Pr[\text{Coll}(q, N)] \geq (q(q-1))/4N$. Let us see how we can derive this.

So, let me denote the event NoColl_i to be the event when we made the first i queries, all the hash output values are distinct, that means the values y_1, y_2, y_i are all distinct. That is the case that means $H(x_1), H(x_2), \dots H(x_i)$, they are all distinct and no collision has occurred and that is the event NoColl_i . It is easy to see that since we are making q number of queries, then the event $\text{NoColl}_q = \overline{\text{Coll}(q, N)}$ because if the event NoColl_q occurs that means all your y values y_1, y_2, y_q are distinct and that means there exists no collision, which I obtained by running by a generalized algorithm.

So now it is easy to see that by applying simple rules of probability, we obtain the probability that event $\Pr[\text{NoColl}_q] = \Pr[\text{NoColl}_1] * \Pr[\text{NoColl}_2 | \text{NoColl}_1].. \Pr[\text{NoColl}_q | \text{NoColl}_{q-1}]$. That means the first $q-1$ queries give you a distinct output, what is the probability that event when you make the q^{th} query, there exists still no collision. If I multiply all these probabilities, I obtain the probability of the event NoColl_q . So let us calculate each of the probabilities that are there in your right hand side. So it is easy to see clearly that the event NoColl_1 occurs with probability 1 because when you are making your first query, definitely no collision has occurred. That means y_1 is distinct. So I can simply turn off this, ignore this probability because that is 1.

Now let us compute the i^{th} term in the expression on your right-hand side. Namely, let us compute the probability $\Pr[\text{NoColl}_{i+1} | \text{NoColl}_i]$. That means your event is the following: your $y_1 \neq y_2 \dots \neq y_i$ are all distinct that is given to you and then when you have made the query for x_{i+1} , you have obtained y_{i+1} and we want to analyze what is the probability that is y_{i+1} is different from all the values y_1 to y_i .

It is not difficult to see that this probability is nothing but $1-i/N$, why? Because $1-i/N$ denotes the probability with which this $i+1^{\text{th}}$ value could be same as at least one of these i values and if you subtract that quantity from 1 that gives you the required probability, and I

can always replace this $1 - i/N$ by $e^{-i/N}$ and this comes from your basic inequality from the fact that $1-x$ can be always upper bounded by e^{-x} for all x . So now I know the value of each of the terms in my right hand side that is what I have derived.

So what I have to do is I have to just substitute here. So, I obtained that the probability that the event NoColl_q occurs is the product of these entities and if I take the product in the exponent, basically I end up doing the summation in the exponent. So, I obtained that even NoColl_i occurs with this much probability, and if I take the summation inside, I obtain that this is less than equal to $e^{-q^*(q-1)/2N}$ and then I can finally replace it by this inequality.

This is because $q^*(q-1)/2N$ is less than N because I am making the assumption that q is upper bounded by a square root of $2N$. If that is the case, then indeed $q^*(q-1)/2N$ is less than N , and if that is the case, then I can use the fact that e to the power $-x$ is less than equal to $(1-x)/2$. So by making using all these facts and doing the substitutions and solving the inequalities, I obtained that the event NoColl occurs with this much probability, but that is not our goal.

Our goal is to compute the probability of the event $\text{Coll}(q, N)$ to occur and since the event called NoColl_q and the complimentary event $\text{Coll}(q, N)$ are related by this. We simply obtained that the probability of the event $\text{Coll}(q, N)$ to occur is at least $q^*(q-1)/4N$.

(Refer Slide Time: 10:38)

Implications of Birthday Attack

□ Let $H: \{0, 1\}^* \Rightarrow \mathcal{Y}$ be a **random function**, where $|\mathcal{Y}| = N$.

If $q \leq \sqrt{2N}$ then $\Pr[\text{Coll}(q, N)] \geq \frac{q(q-1)}{4N}$ }

□ Let H **map people to their birthday** ---- $N = 365$, assuming **non-leap years**

$\Pr[\text{Coll}(27, 365)] \approx 0.5$

□ Let $H: \{0, 1\}^* \Rightarrow \{0, 1\}^\ell$ be a hash function --- $N = 2^\ell$

128
2
1 > 256

❖ Let $q = 2^{\frac{\ell+1}{2}}$ ✓

❖ Then $\Pr[\text{Coll}(q, N)]$ is a **constant**

❖ ℓ has to be **significantly large** so that $2^{\frac{\ell+1}{2}}$ hash value computations becomes **impractical**

➤ Large value of ℓ **only a necessary condition** for a CRHF

So that is the lower bound we have established here assuming that your function H is a random function and we have evaluated the function H at q distinct x values, and we have

established that the probability of the collision is at least as much. Now you might be wondering that why this generic algorithm for finding the collision is called or is given a fancy name birthday attack. Basically, this is due to the fact that whatever analysis we have done you can model it as of following computer science problem.

Imagine that you have a set of q people sitting in a room who are the random set of people, and each of them has a birthday. So, you can imagine that the function H here is nothing but it is a function which maps the people to their birthdays and we assume that the birthdays of the people fall in a non-leap year. So the number of candidate's birthdays which those people could have is 365 possible values. So, you have q number of people, say person 1, person 2, and person q and their birthdays are $H(P_1)$, $H(P_2)$, $H(P_3)$ and so on.

We want to find out what is the probability that among those q people, there exist at least 2 people who have the same birthday. I am not interested in the year, even I am not arguing the day. I do not require that they should have said same year of birth as well. I am just interested whether they have the same birthday or not and you can imagine that the function H here is mapping those people to birthdays.

So, it turns out that whatever analysis we have done here in the context of hash function, it could be carried out even in the context of this birthday problem and it could be proved that if there are 27 random people in the room, then the chance that at least 2 of them have the same birthday is approximately 50%, so which is a good enough probability. You do not require a large number of people to ensure that with good probability 2 people have the same birthday.

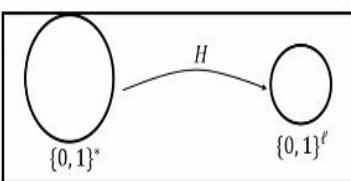
So coming back to the context of cryptographic hash functions, so what exactly are the implications of this birthday attack? It says that imagine that your co-domain space is 2^l . That means your hash outputs could be any string of l bits, then what is the analysis of the birthday attack basically says is that if you evaluate your hash function at these many number of distinct inputs, namely 2^{l+1} over 2 number of distinct inputs. If I substitute this value of q here in this expression and the value of n to be 2^l , then I find that the probability of the collision turns out to be a constant independent of the size of l and a constant success probability is a good enough probability for an adversary to find collision in your underlying hash function in a polynomial time.

That means your value of l has to be significantly large because if you ensure that your value of l is significantly large that means to make these many number of queries, the adversary has to do an impractical amount of computation and that will rule out the implications of birthday attack. Theoretically, the implications are still there. It is only that we are going to operate our underlying hash with such a large value of l that carrying out these many number of hash evaluations is going to be impractical, and that is why a necessary condition for obtaining a collision-resistant hash function is that you should have a large value of l .

Because if l is small, then just by making these many number of queries, adversary could come up with a collision with almost a constant success probability. So that is why the minimum value of l which is recommended for current practical instantiations of the hash functions is 256. Because if you set l to be 256, then basically adversary has to do computations of order 2^{128} to obtain collisions with a constant Success probability, but 2^{128} computations is really a huge amount of computation.

(Refer Slide Time: 14:52)

Small Space Birthday Attacks for Finding Collisions



Collision-finding algorithm:

- ❖ Evaluate H at q **distinct inputs** x_1, \dots, x_q
- ❖ **Output** x_i, x_j if $H(x_i) = H(x_j)$
- ❖ **Space complexity** --- $\mathcal{O}(q)$

❑ **Claim:** Consider the sequence of values y_1, \dots, y_q , where $y_i = H(y_{i-1})$. If $y_i = y_j$, for some $1 \leq i < j \leq q$, then there is an $i < j$, such that $y_i = y_{2i}$

$H(y_{i-1})$ $H(y_{2i-1})$
 y_{i-1} y_{2i-1}

- ❖ **Modified Birthday attack** --- need to **store only 2 hash values** in each iteration
- **Stage I:** Keep computing the values y_1, \dots, y_q and **stop** when $y_i = y_{2i}$ holds
- **Stage II:** "Backtrack" and find the actual collision

So now let us discuss another type of birthday attack, which we call as a small space birthday attack for finding the collision. The idea here remains the same. So, remember in the generalized collision finding algorithm, we evaluated the hash function at q number of distinct inputs, and we had to store all those hash values because when we are comparing the x input and your hash values to find out whether there exist a collision or not, we need all the (x_i, x_j) pairs and their corresponding hash values. It turns out that the space complexity here is of order q .

So, if you consider in the modern computer systems, time is not an issue, the processing speed is not an issue, day by day the processing speed is increasing. What matters is the space complexity because space is really a very critical resource here and that is why what will be interesting here is whether we can carry out the birthday attack or the generalized birthday attack that we had seen earlier where the space complexity is not $O(q)$, but rather say it is a constant?

Now, you might be wondering that how can just by storing constant number of values, we can still perform an attack which is similar to the generalized attack that we had seen in the context of birthday attack. So, the idea behind this constant space complexity or a small space complexity birthday attack is as follows: imagine you have a sequence of values y_1 to y_q which are basically obtained by a sequence of a chain of hash evaluation.

So, you will start with say a distinct or random y_1 and then y_2 is nothing but hash of y_1 and then from this y_2 I obtain y_3 which is same as say hash of y_2 , which in turn is same as hash of hash of y_1 and so on. So, you can imagine that I have a chain like this and each subsequent y value is related to the previous y value by the hash function H . So, that is how I have computed these y values.

Now, we can prove that if you have 2 indices I and J such that your y_I value and y_J values are same. That means you have this chain of values y_1, y_2 , and say you have 2 indices intermediate indices y_I and y_J and you have say done q number of such y computations and you have say 2 such indices y_I and y_J such that y_I and y_J are same. Then it means that there exist at least one index y_i or one y value which I denote by y_i such that y_i and $y_{y_{2i}}$ will be same that constitutes a collision for you. So that is a claim here okay.

So, I am not going into the details of the proofs of this claim, but you have to believe me that if indeed we have computed y values like this and if we have this condition to be there to be true, then this condition holds as well. That gives you an idea of how to find a collision with a space complexity of constant. So, what you have to do is you have to perform the same analysis that you have done for the previous case, but now you just need to store 2 hash values.

So, what you have to do is in each iteration, you have to compute the next y value. So you have to go from y_i to y_{i+1} and at the same time, you have to keep a track of y_{2i} as well. As soon as you obtain an index i such that y_i and y_{2i} are same. That means, there exists some collision in the sequence of y values that you have computed till y_{2i} . So, what you have to do is basically then you have to do a backtracking and find out the exact collision.

So, I stress that is this claim does not give you the guarantee that y_i and y_{2i} is exactly the collision because y_i is obtained from the value $H(y_{i-1})$ and y_{2i} is obtained by evaluating y_{2i-1} . So, it is not guaranteed that y_{i-1} and y_{2i-1} constitutes a collision. That is what you have to basically check by doing the backtracking. So, the interesting part here is that both for stage 1 as well as for stage 2, we just need to store 2 hash values.

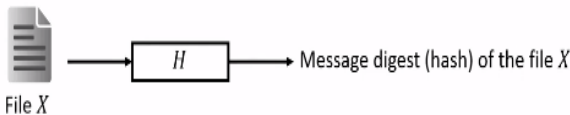
So, in stage 1, we can go in the forward direction and stop as soon as this condition holds and which will become true with very high probability, we can prove that, and if indeed this condition becomes true, we stop there and then we do a backtracking again by just keeping track of 2 hash values and we will end up finding the exact collision in the chain of values that we have got. The probability analysis more or less remains the same. So that is the idea behind a small space birthday attacks for finding collisions.

(Refer Slide Time: 20:23)

Additional Applications of a CRHF

□ Let $H: \{0, 1\}^* \Rightarrow \{0, 1\}^l$ be a CRHF

- ❖ Hash of a file can serve as its **short unique identifier** (irrespective of the file size)



```

graph LR
    A[File X] --> B[H]
    B --> C[Message digest (hash) of the file X]
            
```

- ❖ **Collision-resistance** of H implies finding **different files** with the same digest is **infeasible**

□ Several applications of the above concept

- ❖ **Virus Fingerprinting**
 - Virus scanners store the **hashes of known viruses**
 - When an **email attachment or an application is downloaded**, its hash is compared with the known hashes in the table to identify potential viruses

So now let us discuss some additional applications of collision-resistant hash function. So, the major application that we had discussed in the last lecture was constructing message authentication codes for arbitrary length inputs, but in this lecture, we will see some other applications as well.

So, imagine you are given a collision-resistant hash function and the one of the main ideas that we can use in several applications involving the hash function is that the hash of a file can serve as its short unique identifier where the size of the identifier will be fixed, say l bits and its size would be l bits irrespective of what file you are feeding as an input to the hash value. It could be a small file, it could be a large file, I do not care. The digest of the file will be taken as an identifier and since we are assuming that your underlying hash function is collision resistant that means in practical amount of time, it will be very difficult to come up with 2 files, say X and X' such that both of them give you the same digest.

Because if that is the case, then this whole idea would not work, but since I am assuming that my underlying hash function is a collision-resistant hash function, then it means that in practical amount of time, it is very difficult to come up with 2 two files X and X' having the same identifier or the same message digest and it turns out that this small concept you can use in several real world applications. So let us discuss a few of them.

The first application that where we can use this idea is that of virus fingerprinting. So what we do here is that the anti-viruses or virus scanner stores the hashes of known viruses in their database and whenever we have an email attachment or we download an email application, and if we want to check whether the downloaded attachment or the download application has a virus or not.

What we do is the virus scanner basically computes the hash of the downloaded attachment or the application and then it matches the hash with the known hashes of the viruses, which it has already stored in its database. If there is a match, then an error message is given to us that it suspect data attachment that we have obtained or the application that we have downloaded contains potential virus. Notice that it might be possible that you have an attachment which is a genuine attachment, but unfortunately its hash matches the hash of one of the known viruses. In that case, even for a genuine attachment, you might get an error message, but that implies that you are getting a collision, or you are finding a collision in a feasible amount of time.

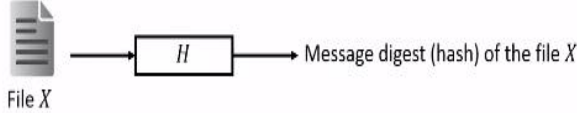
But since I am assuming that my underlying hash function is collision resistant, if at all an error message is given to me that means at very high probability indeed the hash of the attachment that I am downloading is containing viruses.

(Refer Slide Time: 23:20)

Additional Applications of a CRHF

☐ Let $H: \{0, 1\}^* \rightarrow \{0, 1\}^\ell$ be a CRHF

❖ Hash of a file can serve as its **short unique identifier** (irrespective of the file size)



```
graph LR;
    FX[File X] --> H[H];
    H --> MD[Message digest (hash) of the file X];
```

❖ **Collision-resistance** of H implies finding **different files** with the same digest is **infeasible**

☐ Several applications of the above concept

❖ **Deduplication**

➤ **Eliminate duplicate copies** of the same data

➤ Highly relevant in the context of **cloud storage**, where multiple users rely on a single cloud storage to store their data

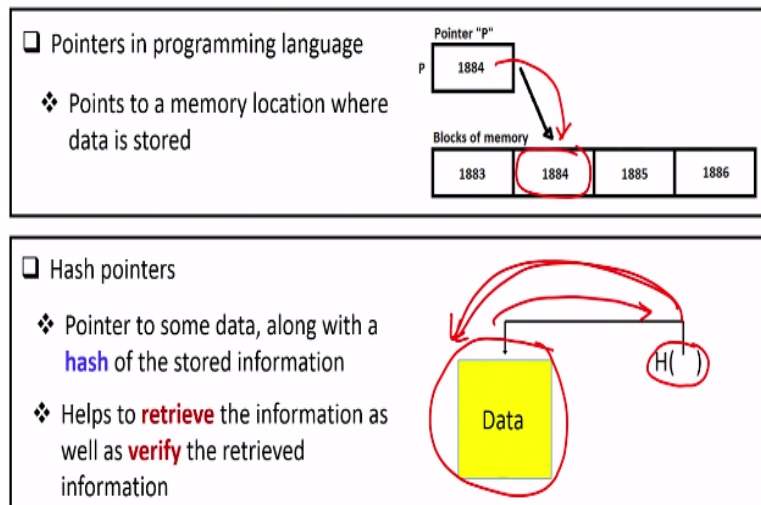
The second application where we can use this idea is that of the duplication and here the idea is that we want to eliminate duplicate copies of the same data. So, imagine that we are in a cloud storage scenario where say we have several users, user₁, user₂ and say n number of users and each of them are hiding a common cloud storage for uploading their data. Now, since the users are independent of each other, it might be possible that user₁ and user₂ end up uploading a same file.

If the cloud storage naively gives or stores both the copies of X, then that will be a wastage of space. So what can be an interesting or smart solution will be every time a user tries to upload a file, what the cloud storage can do is it can compute a hash of the file for which the request has come and it can compare that hash value with all the hashes of the previous files that it has stored.

If the hash value matches that means that the file which has been requested to be uploaded is already there in the cloud storage, so no need to again create a fresh copy of that file. So that is the idea of deduplication here, and again, we are using the idea of hashing the file and using the hash of the file as its identifier.

(Refer Slide Time: 24:38)

Hash Pointers



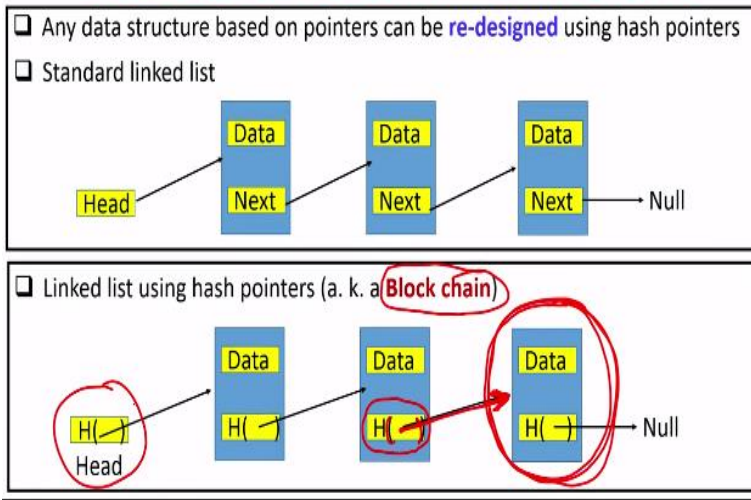
Now let us see some other applications of hash functions and this is what we call as hash pointers. So I am assuming that all of you are familiar with pointers in programming language. Basically, pointers are special type of variable or you can imagine that basically pointer variables points to a memory location where some data is stored. So if say 1884 is a memory location, and if I have a pointer variable, then in that pointer variable the address 1884 is stored.

So by following the address you can or by following this pointer, you can come to the location 1884 and see what exactly is stored. Now this hash pointer is almost same as a pointer in spirit, but apart from pointing to the data, this pointer also stores the hash of the value that is stored in that location. That means if this is some data which is stored in some arbitrary location, and if I say that this is your hash pointer, then this hash pointer will have a pointer to the data as well as the hash of the data that is stored in that location.

The advantage of this hash pointer is that by following this pointer, not only you can retrieve the data, so if you follow the pointer you can retrieve the data, and once the data is retrieved if you want to verify whether you have retrieved the right data or not, you can again hash it and compare it with the hash value which is stored along with this point. So that is advantage or the extra property which is available with the hash pointer compared to the traditional pointers.

(Refer Slide Time: 26:10)

Data Structures Using Hash Pointers



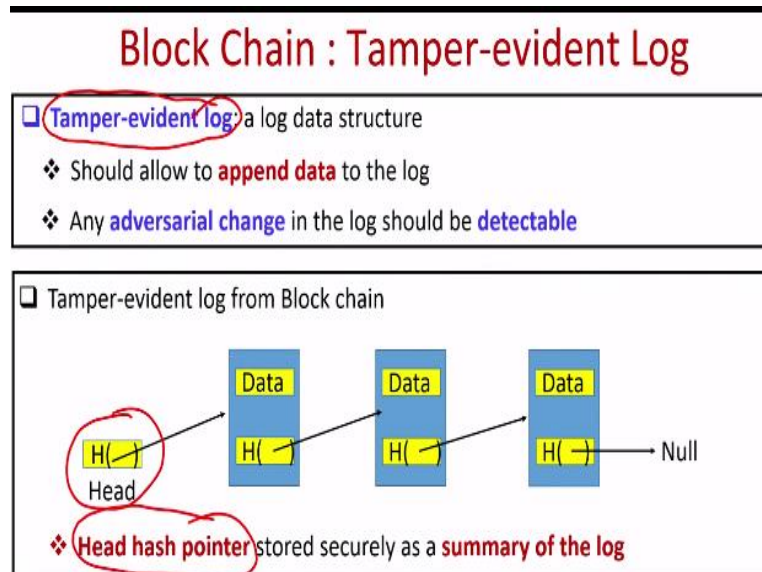
Now it turns out that any data structure which we can design using the standard pointers in programming language, all of them can be redesigned by replacing the standard pointers using hash pointers. So for instance, this is your standard linked list where you have a collection of nodes and each node has some sequence of data of which we call as data block and a pointer to the next block, like this, so this is your last block, and since there is nothing after it, that is why the next pointer for it is set to be null.

We have the previous block where we have the data and a pointer pointing to the next block and so on and like that we have the head pointer. So that is a standard linked list here, which I guess all of you might be aware of. What we now do is that we take the same linked list, but replace all the pointers by hash pointers. So, we have now a sequence of blocks here, each block will have its own data content and along with that instead of a regular pointer, we have a hash pointer, which will point to the block after it along with the hash of the data as well as the hash pointer stored here.

So that means if I consider this hash pointer, it will point to the next node and when I say node, node means node as a whole that means the data concatenated with the hash pointer stored there. So this hash pointer will be pointing to this data block and along with that a hash of this whole block whole binary string, whole binary content that is kept here will be stored in this hash pointer. Now it turns out as soon as I take the standard linked list and replace all the pointers by hash pointers, I obtain a very nice data structure, which has a very fancy name in this today's world, namely Block chain.

So, if someone ask you what exactly is a Block chain? Block chain is nothing but a linear linked list where all the pointers are replaced by hash pointers. That means you have a sequence of block, each block has its own data part and short hash or the short summary of the entire content of the block after it and so on and like that you have a short summary of the entire Block chain or the entire linked list, which is nothing but the hash value of the head data block.

(Refer Slide Time: 28:44)

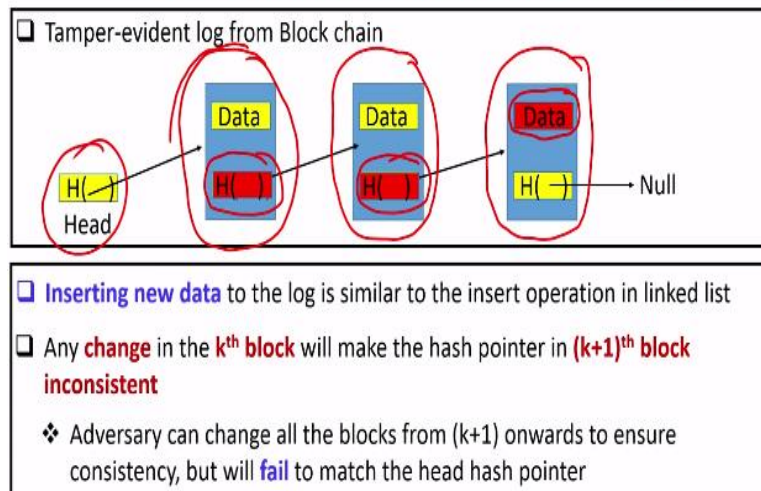


So, this Block chain, it gives you a very nice data structure. Namely, it helps you to instantiate what we call a tamper-evident log. What exactly is a tamper-evident log? It is a log data structure, namely it will store lots of information and it should be a dynamic data structure in the sense it should allow you to append data to the log. That means whatever existing log you have, you can append data to it. And apart from that any change in that log or the existing log should be deductible.

So let us see whether we can instantiate this tamper-evident log using Block chain. So it turns out that indeed, we can use the Block chain to instantiate tamper-evident log and the idea here is that we can use the head hash pointer as a short summary for the entire log. That means I do not need to store the entire log. It is enough for me if I just store the head hash pointer. By just storing the head hash pointer, I can detect later on whether any tampering has occurred in the existing log or not.

(Refer Slide Time: 29:49)

Block Chain : Tamper-evident Log



Let us see how exactly the things go here. So, if at all you want to insert a new data, that means suppose if this is an existing Block chain and if you want to insert a new data block here or a new node here, that is very simple. You create the data, whatever data you want to store and along with that, you create a hash pointer here, which will point now to the previous head block here, and once you do that, you create a hash of this entire thing and that will be the hash pointer, which will be now stored in the summary of this tamper-evident log.

So inserting a new data here is exactly the same as the insertion operation in the standard linked list, but now the interesting part here is that any change which adversary tries to make in an existing log with very high probability, it will be detected. So imagine, for instance, the adversary tries to change the contents of the data block in the third node here. So at the moment, I am assuming that you do not have the existing Block chain. You just have the head hash pointer with you.

Now you ask someone that you please give me the content of the third node here or you give me the entire Block chain. So suppose if the request goes to the adversary, then what the adversary can try to do is that instead of giving you a genuine copy of the whole log or the whole Block chain, he might now try to insert his own data or he might want to change the existing data and so on and now he might want to give you the changed Block chain.

So the idea here is that if at all he tries to do that, he will be detected with very high probability assuming that your underlying hash function is a collision-resistant hash function. Let us see how it happens. So imagine it changes the contents of the data block in the third

As soon as he changes the hash value in the second node, the node 2 as a whole it is binary content becomes different. To make it consistent, what he has to do, the adversary has to now change the hash pointer which is stored in the first node and as soon as the adversary changes the hash pointer in the first node, the binary contents of the first node will become different, and to make it consistent basically adversary now has to change the contents of the head hash pointer as well.

That is why the Block chain is a very popular database because you can keep on adding data to the existing log and the whole log can be summarized by just storing a small hash value.

Merkle Trees: Binary Tree with Hash Pointers

Full binary tree (2^k leaves)
based on hash pointer

Tampering of any data block can be detected

Let us see the last application of hash function for today's lecture. This is a very nice application which we call this Merkle tree and what basically Merkle trees are they are full

binary trees with hash pointers. So all of you, I am sure you know what exactly is a full binary tree. So you have say 2^k number of leaves and at each layer, you have a node with both its left child as well as right child being present here. So what we are now doing is that instead of implementing the binary k with standard pointers, we now replace those standard pointers using hash pointers.

So we imagine that we have say 2^k number of data contents or data blocks here. So in this example, I am taking say 8 number of blocks and the way we construct a Merkle tree is as follows. So we have at the leaves the data blocks present, and then we go to the one layer up, where we have the hash pointers. So this hash pointer basically contains the hash of the data and this hash pointers basically contains the hash of this data and so on. Now, what we do is since this hash values are also binary strings, we can take this binary string concatenated with this binary string and hash it and go one layer up.

So, this hash value basically is a hash value of concatenation of these 2 hash values. In the same way, this hash value is basically hash value of the concatenation of these two has values and so on and we go one layer up and like this we keep on going till we obtain the hash root pointer and that hash root pointer or the hash value which is stored at the root basically is a short summary of the entire tree, and we require that in this Merkle tree, the number of leaves node should be a power of 2.

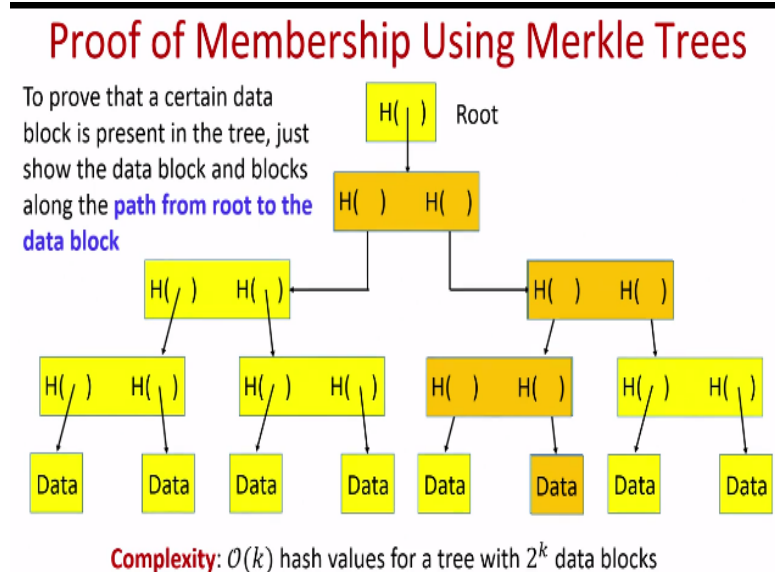
If that is not the case, then we append dummy nodes to ensure that the number of leaf nodes is indeed some power of 2 and that is required for the overall application here. So, as it was the case for Block chain or the hash pointer based linked list, it turns out that any change or any tampering of data block can be easily detected if someone is maintaining up to date copy of the hash root pointer. That means, imagine I as a user, I am not maintaining the whole tree, I am just having hash root pointer with me.

Suppose at a later point of time, I ask someone that please give me all the data block contents, then if that user tries to change the data block or the data content and with very high probability it will be detected assuming finding collisions in the underlying hash functions is difficult. So for instance, suppose it tries to change the second data block while giving it back to me. If it changes the second data block, then the hash value stored in the previous tree will

get changed here and as a result this binary string of this node as a whole will become different.

To make it consistent, the adversary has to change this hash value, which will further disturb this whole binary string as a whole. To make it further consistent, the adversary has to change this hash value and this as a whole will make this binary string disturbed and as a result adversary has to change the head hash pointer or the root hash pointer assuming that finding collisions are difficult, but as soon as the adversary tries to change the root hash pointer, it will be detected because that would not match with the root pointer that I am storing with me.

(Refer Slide Time: 36:37)



The interesting part of this Merkle trees is the proof of membership. So what exactly is the scenario here? Imagine that I have stored only the head root pointer with me and I do not have the data blocks. Now suppose later on I ask someone that please give me i^{th} data block. Now the person from whom I am asking the i^{th} data block if he is malicious person, he might try to give me an incorrect i^{th} data block. How do I verify that whether indeed the so called i^{th} data block that he is giving to me is a correct data block or not?

Well, one way of verifying that is that I re-compute the entire tree, but for re-computing the entire tree not only I need the i^{th} data block, but I need all the data blocks which were present in the original Merkle tree because only when I have all the data blocks present with me, I can re-compute the root along given and taking the i^{th} data block that the person has given to me and then verify whether indeed he is giving me the correct data block or not, but it turns out that we do not require all the data blocks.

Namely, 2^k data blocks as a proof to verify whether the retrieved i^{th} block is indeed current i^{th} block or not, magically, it turns out that only by giving you k number of data blocks, one data block and k number of intermediate blocks it is suffice for me to re-compute the root here. So basically that acts as a proof for the person who is giving me back the i^{th} block. So if he is giving me back the i^{th} block and he wants to prove that indeed the block that he is giving to me is the current i^{th} block or not.

Basically, what he has to do is that apart from that i^{th} data block, he can give me all the blocks along the path from the root to the data block and just by using the blocks along the path from the root to the data block, which I have actually asked for, I can re-compute back the root pointer or the root hash value and compare it with the root hash value that I have stored with me to verify whether indeed the data block that I have received is correct or not. So let me demonstrate what I am trying to say.

Imagine I have the up to date root pointer or root hash value, and I asked from a person who has actually maintaining a full copy of the Merkle tree that “please give me the sixth block here” and he gives me the sixth block, and if I consider the path from the root to the sixth block, the nodes that are occurring along that path are now highlighted with this highlighted blocks and what basically that person has to do is he can give me the sixth block and along with that he can give me all the binary strings along with this highlighted nodes.

These highlighted values suffice for me to re-compute back the root of the original tree and by re-computing the root of the original tree, I can compare it with the root value that I have stored with me assuming finding collisions in the hash functions are difficult, with very high probability it is ensure that indeed the so called sixth block that the person has given to me is indeed the correct sixth block which was present in the original Merkle tree. So that means, now you can see that the number of values that the person has to give me to prove that indeed he is supplying me the correct i^{th} block is not of order 2^k .

Basically, for each layer of this complete binary tree, he has to just give me one node value. So, at this layer he has to give me the binary contents here, at the second layer he has to give me the binary contents of this node, at the third layer he has to give me the binary contents of

this node, and at the last layer basically he has to give me the i^{th} the data block, in this case the sixth data block.

That means, if you have a Merkle tree where there are 2^k number of data blocks and if you want to fetch only a particular data block i^{th} data block, then the number of binary strings which the person has to give back to me is of order k only, not 2^k and that makes this data structure very powerful. You can prove the membership of certain data blocks in the whole tree without actually supplying the whole tree. Just supplying logarithmic number of information, it suffices for an entity to prove whether a certain data block is present in the tree or not.

So, that brings me to the end of this lecture. Just to summarize, in this lecture we had seen some of the applications of hash functions. We have seen how the hash of a file can or the hash of an entity can serve as its unique identifier assuming finding collisions are difficult, and this concept has got tremendous applications. Major application is in the context of hash pointers, where we can replace all the data structures based on standard pointers by hash pointers.

We had seen how Block chains can be used as our tamper-evident log based on this principle. We had also seen Merkle trees and how Merkle tree helps us for the efficient proof of membership. Thank you!