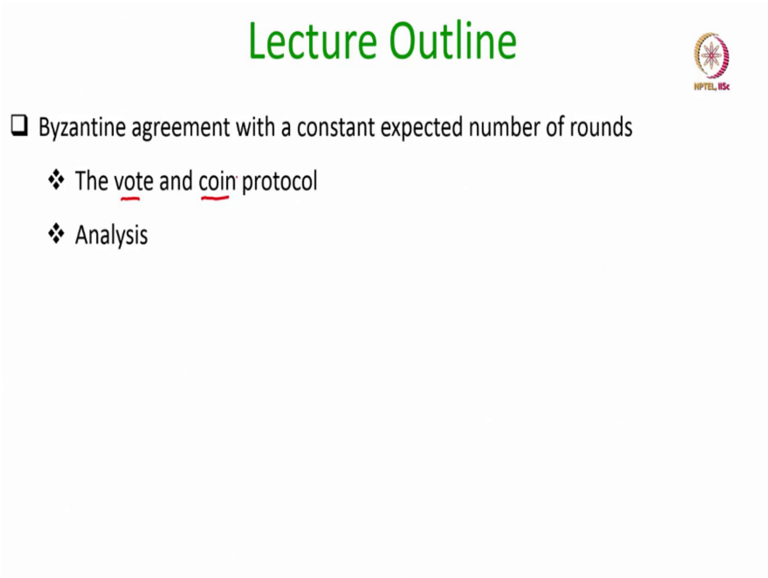
Secure Computation: Part II
Prof. Ashish Choudhury
Department of Computer Science and Engineering
Indian Institute of Science, Bengaluru

Lecture - 13
Randomized Protocol for Byzantine Agreement: Part II

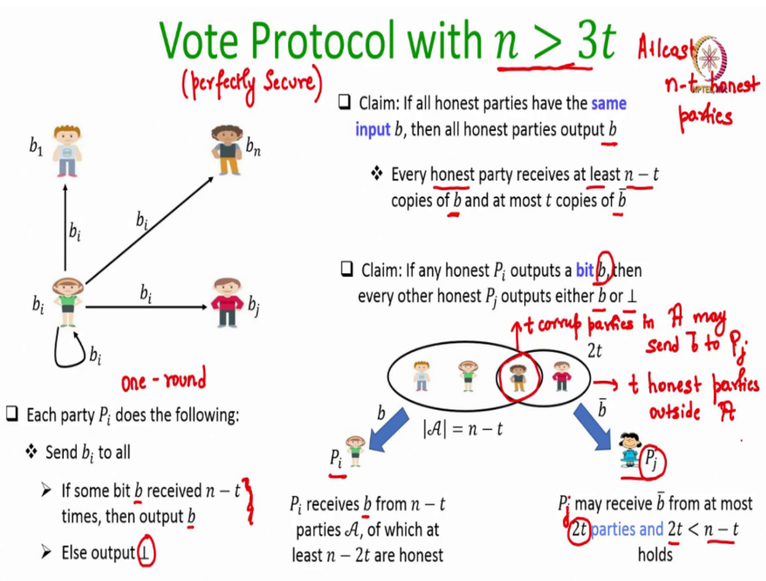
Hello everyone, welcome to this lecture. So, we will continue our discussion regarding randomized protocols for Byzantine Agreement.

(Refer Slide Time: 00:28)



So, we will see the framework due to Benor and Rabin. We will see how we can combine the vote and coin primitives to get a Byzantine agreement protocol with a constant expected number of rounds and will do the analysis of the protocol.

(Refer Slide Time: 00:47)



So, let us first see the instantiation of the vote protocol and this instantiation will be perfectly secure and it will be with $n > 3t$. So, there can be at most t Byzantine corruptions and $n > 3t$. So, the protocol is very simple, it is a 1 round protocol, where each party just has to send it is input to everyone else including itself.

And the output decision is made as follows. Every party P_i checks that it has received a copy of some bit b at least $n - t$ times from $n - t$ different parties, if that is the case then it outputs that bit b otherwise its output is $\perp$. Now, we make some claims regarding this simple vote protocol. So, the first claim is that if all the honest parties have the same input, then everyone outputs that bit b.

And the proof is very simple. If every honest party have the same input b and remember, we have at least $n - t$ honest parties. That means, at the end of the round every honest party will receive at least $n - t$ copies of the bit b and there could be at most t corrupt parties who can send b' as their inputs. And what is the output decision rule?

The output decision rule is that if a bit is received from $n - t$ parties, then output that bit. That is why every party will output b and not b'. And no honest party will output $\perp$ because there is a bit b which is received at least $n - t$ times. The second claim is that if any honest party outputs a bit b, then every other honest party will also output either the same bit b or $\perp$ right. So, there is a difference now between the two claims.

For the first claim statement the hypothesis was that if all the parties have the same input, in that case everyone will output that bit b. The second claim states that it could be possible that even though all the honest parties do not have the same input bit b, it could be possible that one of the honest parties outputs a bit b which could be either 0 or 1.

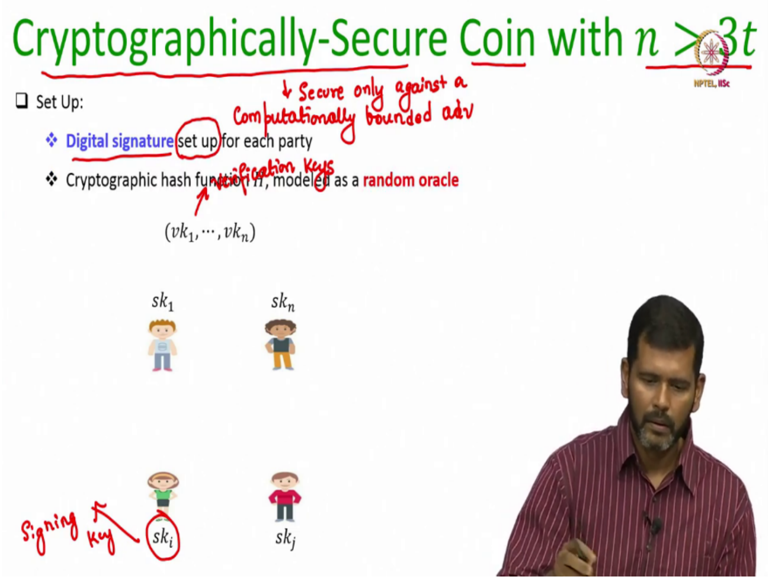
If that is the case then no other honest party will output b', the only output for the other parties could be either the bit b or the value $\perp$. So, again we have proved these claims several times in the past earlier in the context of phase king broadcast protocols and so on. So, the idea behind the proof is as follows. So, imagine there is party P_i and say the party P_i outputs the bit b.

Now, since P_i outputs the bit b; that means, in the protocol P_i would have received the bit b from a set $\mathcal{A}$ of $n - t$ parties. And among those $n - t$ parties at least $n - 2t$ are guaranteed to be honest. Now those $n - 2t$ honest parties in the set $\mathcal{A}$ will also send b as their input to every other honest party P_j. So, consider another honest party P_j different from P_i and let us see how many copies of b P_j will receive and how many copies of b' P_j might receive and let us see what the possible outputs for P_j could be.

So, the party P_j may receive the complementary bit b' from at most $2t$ parties. Who can be those $3t$ parties? There could be up to t corrupt parties in the set $\mathcal{A}$, t corrupt parties in $\mathcal{A}$ may send b' to P_j because they are corrupt. They can send b to one honest party and they can send b' to another set of honest parties and there could be up to t honest parties outside $\mathcal{A}$.

So, all together there could be at most $2t$ parties who may send a bit b' to the honest party P_j. Now what is the output decision rule? Well P_j would have output b' provided it would have received $n - t$ copies of b', but $2t$ is strictly less than $n - t$ because we are working with the condition $n > 3t$. That means, if at all P_j outputs a bit it must be b it cannot be b' or otherwise P_j would output $\perp$. So, either P_j would output b or the value $\perp$, but it cannot be b'. So, that is the vote protocol.

(Refer Slide Time: 06:52)

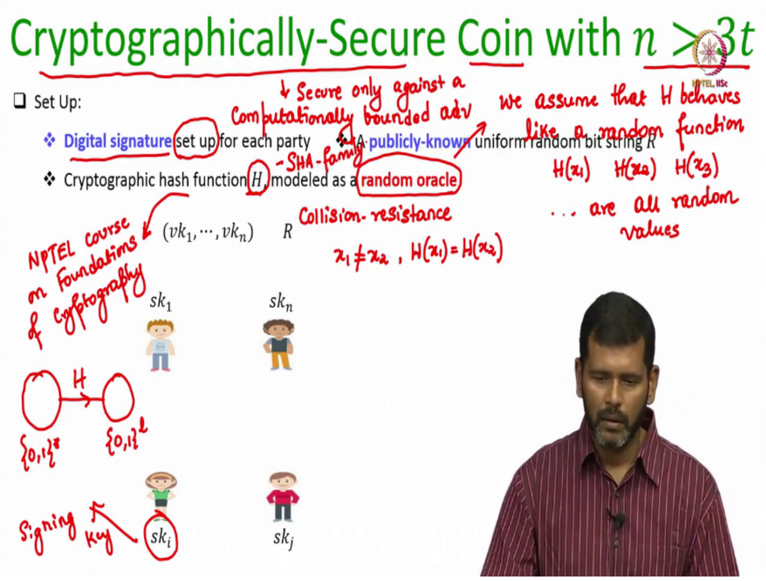


Now, we will give an instantiation of the coin protocol with $n > 3t$ and our instantiation will be cryptographically secure. That means, it will be secure only against a computationally bounded adversary. As a result of that when we will apply this instantiation of coin protocol in the framework of Rabin and Benor, the resultant be a protocol will be cryptographically secure.

Later on, once we discuss advanced primitives advanced tools, we will see an instantiation of the coin protocol which is perfectly secure. So, since we are assuming here a computationally bounded adversary whose running time is polynomial time, we are free to use cryptographic tools. So, we will assume a digital signature setup for every party similar to what we have done in the Dolev strong protocol, and the setup will be that every party P_i will have its own signing key.

And the verification keys of all the parties will be publicly known, this will be a one-time setup which can be used for polynomially many instances of the coin protocol.

(Refer Slide Time: 09:04)



Apart from that we will apart from this setup we will also use a cryptographically secure hash function say H, which is modeled as a random oracle. So, I am assuming here that all of you know what a cryptographically secure hash function is. If you are not aware of what is a cryptographically secure hash function, you can refer to any standard text on cryptography or you can also refer to my NPTEL course on foundations of cryptography. Basically, a hash function is a function which takes inputs of any size and gives you outputs of some fixed size.

And there are many security properties which we require from the hash function, the primary being the primary security property that we require from the hash function is the collision resistance property. Namely it should be difficult to come up with two different inputs x_1 and x_2 which are not same, but their hash values are same, that should be difficult. Even though there are multiple such x_1, x_2 in the domain, because our domain could be infinite, but co domain is finite.

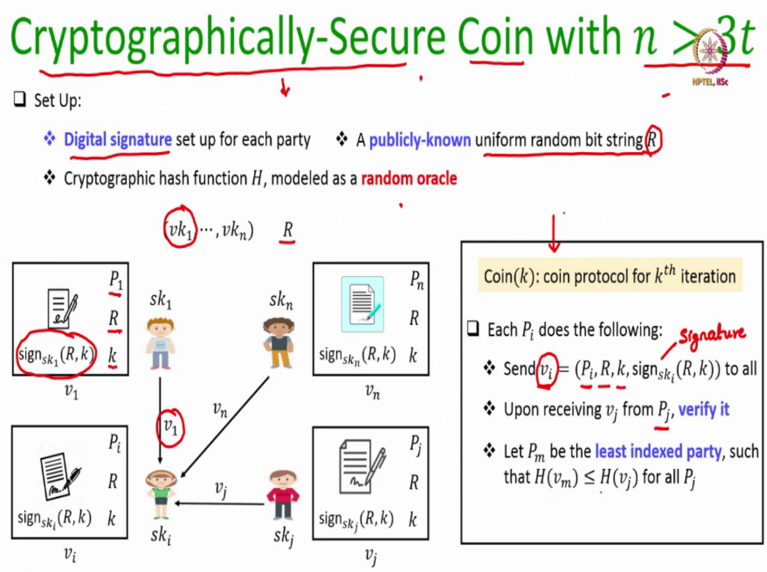
So, from the pigeonhole principle it is straight forward to conclude that there will be multiple such pairs of values x_1, x_2 which are different, but which have the same hash value. But collision resistance demands that it should be difficult to come up or identify or find such pairs in polynomial amount of time. When I say that we are modeling the hash function as a random oracle that also means we are making a very strong assumption regarding the property of the hash function random oracle. Here, we assume that H behaves like a random function, a true random function.

That means hash of x_1 hash of x_2 are all independent values, are all random values, and they are unpredictable. That means, if I am the adversary and if I know the description of the hash function, but I do not know the input for the hash function beforehand then for me the output of the hash function on that input is unpredictable.

So, there are several practical instantiations of hash functions available which you can use by instantiating this coin protocol. So, you can use the SHA family of hash functions. So, that is the setup, a digital signature setup for every party and a hash function publicly known which is treated like a random oracle.

And apart from that, as part of the setup, a publicly known uniformly random string is also available to the parties. That is also a one-time setup which can be used for polynomially many invocations instances of the coin protocol.

(Refer Slide Time: 13:29)



Now, suppose we want to instantiate the coin primitive coin protocol for the kth iteration, where k is a input parameter here. And looking ahead, when we will be using this coin protocol with the vote protocol in the framework of a Benor and Rabin recall that in that framework there are several iterations; where in each iteration, there is one instance of the coin primitive and 2 instances of the vote primitive. So, you can imagine that if we are instantiating the coin primitive during the kth iteration this will be the code which is going to be executed.

So, each party P_i during the coin protocol for the kth iteration will do the following, it will send the value v_i a tuple of values to everyone. So, what exactly this tuple v_i consists of? It consists of the identity of the party, the value of the random string, the iteration number and the signature of the ith party on the string r followed by k. If P_i is an honest party it will send this value v_i this tuple v_i identically to everyone, but if P_i is a corrupt party it may not follow the protocol.

It may send v_i to one set of parties it may send another v_i to another set of honest parties or it may not send any v_i at all, so it can behave in any arbitrary fashion. Now, once every party sends its tuple v_i to every other party, what the party P_i does is the following. It would have received the tuple v_j from many parties from at least all the honest parties, it verifies whether the tuple v_j is correct or not is as per the protocol. And how it can verify this?

So, suppose for instance it has received this tuple v_1. Now v_1 is supposed to consist of the identity of the parties it checks that it should have the value of the random string anyhow that is publicly known. So, P_i can verify that it should check the iteration number that is also a public information.

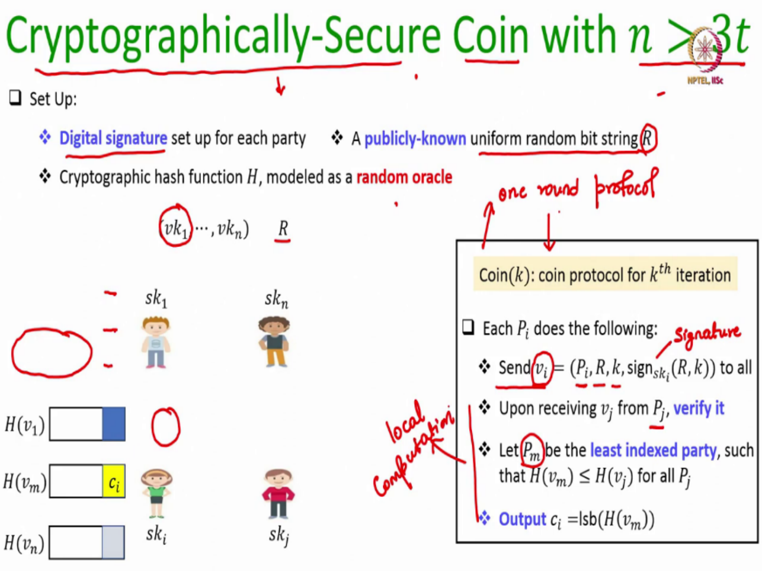
And what is the only thing which P_i must check separately is the signature and that also can be verified, because the corresponding verification key for the party P_1 is publicly available. So, P_i whenever it receives a tuple v_j from any party P_j, can verify whether it is correct or not. If it is correct then it keeps it otherwise, it simply considers that P_i has not sent any value P_j has not sent any value or imagine that it has sent some default message. Now let P_m be the least indexed party such that among all the tuples which are received by P_i the hash value of the mth party's tuple is the least one.

So, again what we are doing here is that we are asking the party P_i to compute the hash value of all the v values that it has received. Now remember H is modeled as a random function.

So, it could be possible that there are 2 tuples which P_i has received which results in the same hash value. So, what P_i is doing is it is looking for the least indexed party whose hash value is strictly less than equal to whose hash value is less than equal to the hash value of the tuples of the other parties.

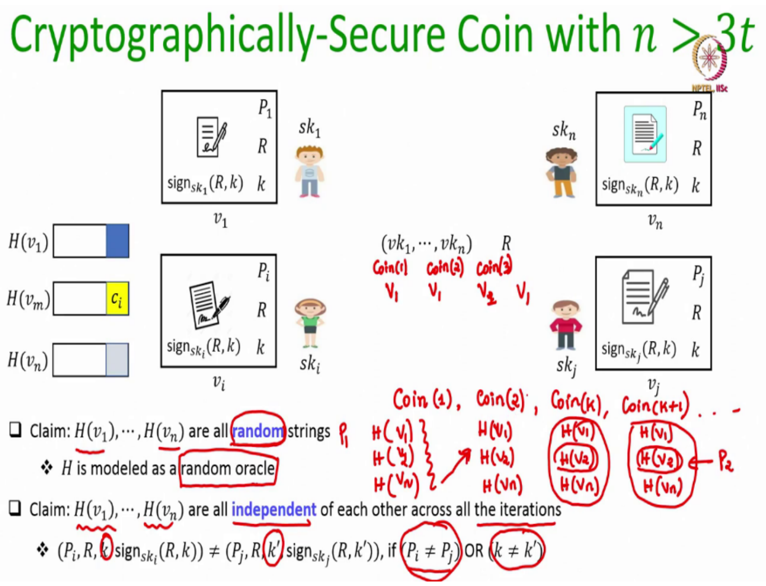
So, imagine P_m is the least such indexed party. Now what is the output for the party P_i? It simply outputs the LSB of the hash value that among. So, there will be the least LSB for all the hash values that it has computed, among those least significant bit P_i will be considering the LSB only for the tuple corresponding to the party P_m which whose hash value is less than the hash values of all the other parties.

(Refer Slide Time: 18:20)



So, again you can see that this coin protocol is a 1 round protocol, because it requires each part to send only the tuple v_i and rest of the steps are local computations. So, that is the simple coin protocol, now we will analyze it.

(Refer Slide Time: 18:50)



So, the first claim here is that if we consider the hash values of all the tuples $v_1, v_2, ..., v_n$ they are random strings from the co domain of the hash function. And this simply comes from the fact that we are assuming the hash function to behave like a random function or a random oracle. The second claim is a slightly stronger claim which states that if we consider the hash values of all the tuples received from the parties across all the instantiations of the coined protocol.

So, remember that coin protocol takes also as input the iteration number k. So, the statement basically says that if we take all possible instantiations of the coin protocol, in every instance of the coin protocol the parties would have computed the tuples $v_1, v_2, ..., v_n$ and their hash values.

Similarly, in the second invocation the parties would have computed the corresponding tuples exchange and computed the hash values. Similarly, during the kth invocation of the coin primitive the parties would have exchanged the corresponding v tuples and computed their hash values and, like that, in every iteration they would have computed the corresponding v tuples and compute exchange them and compute the hash values.

So, what we are claiming here is that if we take the hash values, they will be independent of each other across all the iterations.

That means that if we consider say for instance the invocation k and invocation $k + 1$ of the coin primitive, then none of the hash values which are computed during the kth invocation will have any dependency on the hash values which are computed during the $k + 1$th iteration. And this simply comes because in every iteration the hash values are computed on a tuple which will be which are guaranteed to be different, if the party who is computed them are different.

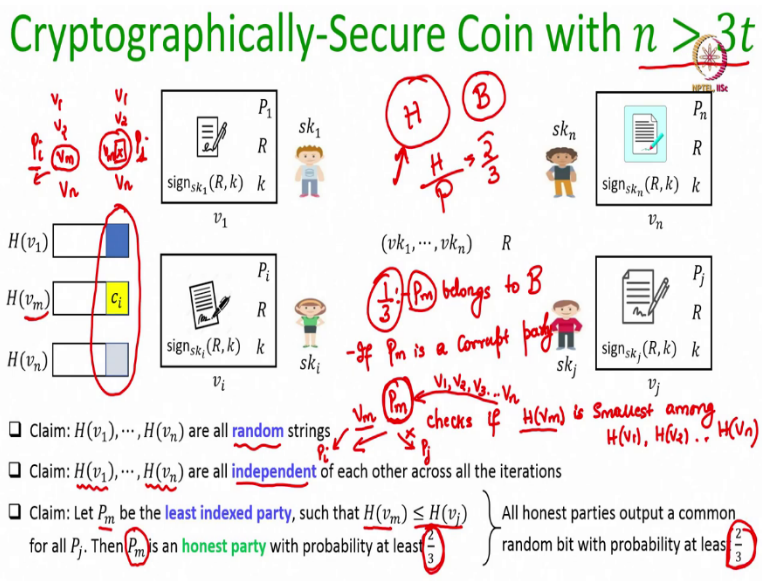
So, for instance if the ith party has computed a tuple and if the jth party has computed the tuple, then either the party indices will be different, or it could be possible that we are talking about say $H\left(v_2\right)$ and $H(v_2)$ in the 2 consecutive instances of the coin primitive. So, both this hash values will be depending upon the party P_2. So, in that case also there will be at least one component in the corresponding v tuple which will be different.

So, for the kth instance of the coin primitive the k component of the tuple will be k; whereas the corresponding component in the $k + 1$th instance will be $k + 1$. So, either the party indices will be different, or the iteration number will be different. And that ensures that the hash values are computed for different tuples across all the iterations. Even if we consider the same party, so that means even if I take say the party number 1 and if I focus on all the v_1 tuples which are computed by P_1 across all the instances of the coin primitive, they are going to be different.

Because the iteration number will vary and since the tuples are different and since we are assuming that the hash function is behaving like a random string. Since we are assuming that the hash function is behaving like a random functions and since the tuples are different across different iterations, the hash function when applied on those different tuples will result in independent outputs. So, there is absolutely no dependency on the hash outputs which are computed during different invocations of the coin primitive.

So, basically through this claim what we are stating here is that if the adversary has seen a set of n hash values during some invocation of the coin primitive, then in the next iteration it cannot cook up its v tuples. So, that it is hash value results in some specific value which adversary would like to get that is not possible.

(Refer Slide Time: 24:42)



So, these are the 2 claims regarding this coin protocol. Now the third claim is that let P_m be the least indexed party such that the hash value of it is tuple is less than or equal to the hash value of all the tuples all other tuples corresponding to the other parties. Then the probability that P_m is an honest party is at least $\frac{2}{3}$ and this comes from the fact that the hash values of all the tuples are random strings and independent of each other.

So that means, now among the n parties if we focus on the set of bad parties and a set of good parties. So, suppose that H is the set of honest parties and B is the set of corrupt parties, then the ratio of the set of honest parties over the set of all parties is at least $\frac{2}{3}$, because we are assuming$n > 3t$. So, $\frac{2}{3}$rd of the parties are honest and $\frac{1}{3}$rd of the parties are corrupt and since the hash values of all the tuples are independent of each other random strings; that means, the LSBs are also random.

So that means, if I focus across the LSBs of all the hash values computed during the coin protocol they are random string. And what we want to analyze here is that what is the probability that the party P_m whose hash value turned out to be smallest belongs to the set H. Well, the probability of that is $\frac{2}{3}$, because there are n number of parties and $\frac{2}{3}$rd of them could be honest. This automatically implies that all honest parties output a common random bit with probability at least $\frac{2}{3}$.

Why is that the case? Because with probability $\frac{2}{3}$ the honest party the party P_m whose hash value turn out to be the smallest one belongs to the set of honest parties. It is only with probability $\frac{1}{2}$ that P_m belongs to the set of bad parties. Now if P_m belongs to the set of bad parties, then this claim then it is not guaranteed that all honest parties have the same output bit, because what the bad party what the party P_m can do is the following.

If P_m is a corrupt party, then it can do the following. So, in the protocol every party would have sent their v tuples to every other party. So, if P_m is a bad party then what it can do is it waits for all the v tuples to reach to P_m and it has not yet computed and it and it has also computed its v tuple say v_m. But right now, it has not sent its v tuple to any other party it is holding it. Now since it has all the n v tuples, what P_m can do? It checks if $H(v_m)$ is the smallest among $H(v_1)$, $H\left(v_2\right)$, ..., $H(v_n)$ it checks that.

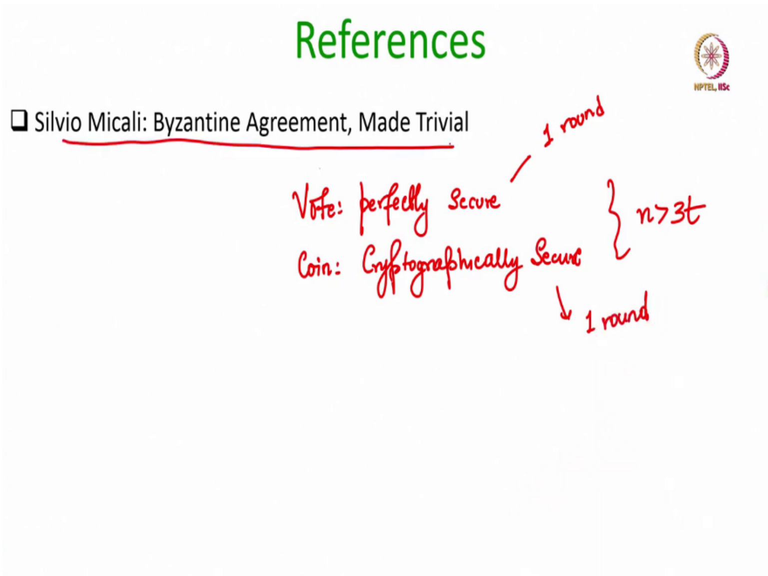
And if it finds that indeed $H(v_m)$ is the smallest among all this hash values then it can decide to do the following, it can send v_m to one honest P_i. But it does not send anything it does not send a tuple v_m to another honest party P_j. This will result in the following scenario, so P_i it will have $v_1, v_2, ..., v_n$ all of them, whereas P_j will have $v_1 v_2$ it would not have v_m and it will have all other tuples. Now for P_i it is the tuple v_m whose hash value will be the least. So, it will output the corresponding LSB, but for P_j it will be some other tuple whose hash value turned out to be the smallest.

Because the tuple v_m whose hash value is supposed to be the smallest among the list is not available with P_j, because the corresponding corrupt party P_m has decided not to send it to P_j; in which case the bit which P_j is going to output will be different from the bit which P_i is going to output. This happens only if P_m is a corrupt party which can happen with probability $\frac{1}{3}$.

Of course if P_m belongs to the set of honest party, honest parties then P_m will not do any such thing it will send the tuple v_m both to P_i as well as to P_j and both P_i and P_j will find that its

v_m whose hash value turns out to be the smallest and both of them will output the same bit. And the probability of that is $\frac{2}{3}$ and that shows the commonness probability for this instantiation of this for the for this instantiation of the coin primitive is $\frac{2}{3}$.

(Refer Slide Time: 31:05)



So, that concludes this lecture. So, we have seen the instantiation for the vote protocol which is perfectly secure. So, the vote protocol is perfectly secure, and our coin protocol is cryptographically secure. Both are designed with $n > 3t$ and both are 1 round protocols. So, vote protocol is also a 1 round protocol coin protocol is also 1 round protocol.

In the next lecture we will see how we club these two protocols two primitives vote and coin in the framework of Rabin and Benor and get the exact byzantine agreement protocol. So, the instantiation of the vote and the coin that I have discussed today is taken from this manuscript.

Thank you.