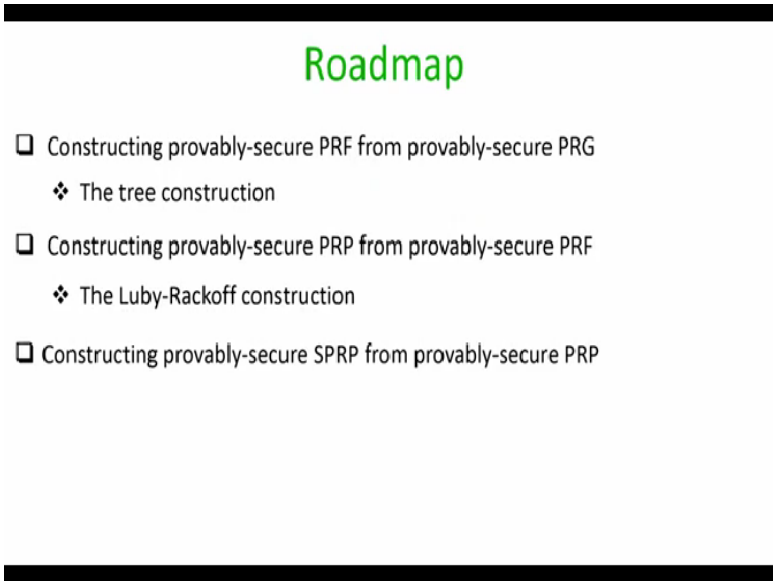
Foundations of Cryptography

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

Lecture-18
Theoretical Constructions of Block Ciphers

Hello everyone, welcome to lecture 17. In this lecture we will see theoretical constructions of block ciphers. Specifically the roadmap for this lecture is as follows.

(Refer Slide Time: 00:37)



So, till last lecture we have seen that if we have a pseudo random function, then we can design candidate CPA secure schemes using most of operations of pseudo random functions. But now the question is how do we actually go about designing the pseudo random function. And it turns out that there are 2 ways of designing pseudo random functions. The first phase, the first class of the constructions are the provably secure constructions, which we are going to discuss here. And they are considered to be theoretical constructions because that is not the way we instantiate pseudo random functions in the real-world protocols.

In the later lectures, we will see the practical constructions namely the constructions which we use in real world to instantiate pseudo random function. However, even though we do not use the so called theoretical instantiations of pseudo random functions, they are very fundamental, they

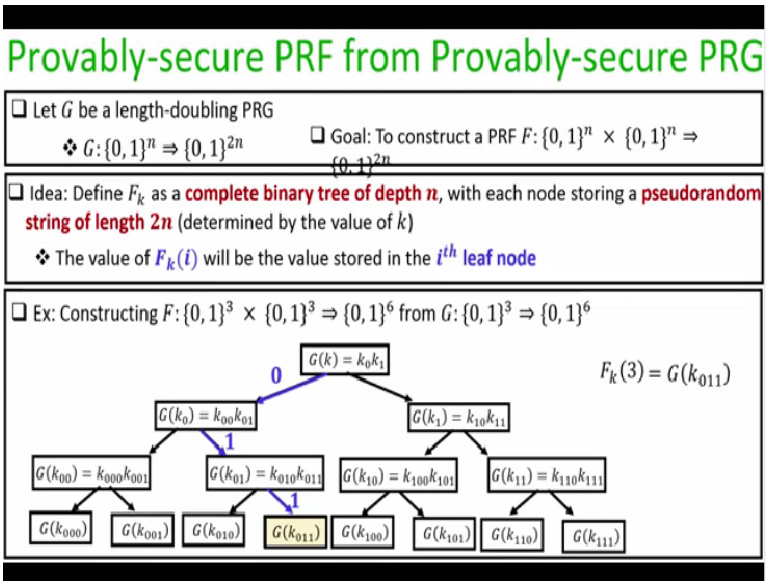
are of fundamental importance in cryptography because mathematically here we show that the constructions that we are going to discuss they can be proved to be secure based on the assumption that one way function exists.

That means we have now mathematical guarantees that the constructions that we are going to see in this lecture they are secure, whereas the so-called practical instantiations which we are going to say in the subsequent lectures, for those constructions we do not have any provable security guarantees that means there is no mathematical proof that indeed those construction satisfies the definitions of pseudo random function, pseudo random permutations and so on.

It is only a belief or an assumption that ever since their discovery, no attacks or no shortcomings have been reported in those constructions. And that is why we believe that those constructions emulate the behavior of a pseudo random function, pseudo random permutations and so on.

So now coming back to this lecture, the roadmap for this lecture is as follows: we will see how to construct provably secure pseudo random functions given provably secure pseudo random generators. Then we will see the constructions of provably secure pseudo random permutations from provably secure pseudo random function. And this construction is also called as Luby-Rackoff construction attributed to the name of their inventors. And then finally, we will see how to construct provably secure strong pseudo random permutations given provably secure pseudo random permutation.

(Refer Slide Time: 02:59)



So, let us do the first thing here. Namely, we will see how if you are given a provably secure pseudo random generator, we will see how to construct provably secure pseudo random permutation. And for the purpose of demonstration, I am assuming that I have a length doubling pseudo random generator $G: \{0,\ 1\}^n \Rightarrow \{0,\ 1\}^{2n}$. And this you can construct in a provably secure way from one-way function using the Goldreich-Levin construction and assuming that hard-core predicate exist.

So, remember in one of our earlier lectures, we had seen provably secure constructions of pseudo random generator, where we first expand the output or the input of the pseudo random generator by 1 bit using hard-core predicate and then we do serial composition of that pseudo random generator polynomial number of times to expand the length of the pseudo random generator by any polynomial amount.

So, I assume that I have such a pseudo random generator, namely a length doubling pseudo random generator and my goal is basically to construct PRF: $F: \{0,\ 1\}^n \times \{0,\ 1\}^n \Rightarrow \{0,\ 1\}^{2n}$. The construction is called a tree construction. And the reason it is called a tree construction is that basically the way we define the keyed function F_k is that we construct a complete binary tree of depth consisting of 2^n leaf nodes

where each leaf node is going to consist of pseudo random string of length 2n bits determined by the value of the underlying key k.

Now, the reason we are going to construct a complete binary tree of depth n is that we are going to have 2^n leaves and each leaf node is basically a value of the function F_k. And this matches with our semantic of the keyed function F_k that we are interested to construct because the block size of my underlying keyed function which I want to construct is n bits. That means, my function $F_k(i)$ namely the range of this i: $0 \leq i \leq 2^n - 1$. So there are 2^n candidate inputs for this function. That is why I am interested to design a tree consisting of 2^n nodes. And ith value or the value of the keyed function F_k on the input i will be basically the pseudo random string which I am going to store at the ith leaf node in this tree.

So, the entire thing boils down to how exactly this tree is going to be defined as a function of my underlying keyed k. So, for the purpose of demonstration, I assume that I have pseudo random generator $G: \{0, 1\}^3 \Rightarrow \{0, 1\}^6$. And using that I have to design a keyed pseudo random function, $F: \{0, 1\}^3 \times \{0, 1\}^3 \Rightarrow \{0, 1\}^6$.

The construction is as follows: so this is your complete binary tree of 8 nodes. So this is your 0th leaf. This is your first leaf and this like this, this is your 7th leaf. And this will denote the strings that we are going to store in each of these respective leaf nodes will denote the value of the function F which we are going to define at their respective inputs. So now let us see what exactly will be the bit strings which are going to be stored in each of this internal nodes and the leaf nodes.

So to begin with at the root of the tree, we are going to store the value k_0k_1 which is a bit string of length 6 bits, and which is generated by actually invoking the pseudo random generator on the key k. So, remember k is basically the key of the pseudo random function which I am interested to design, but now that key I am using as the seed for the pseudo random generator. And since my pseudo random generator expands the seed and gives me an output which is twice the size of

the input, I will obtain a pseudo random output which I can parse as 2 blocks of 3 bits, 3 bits each.

Now in my left-hand side note, which is that with the left child of this root, I basically stored the output of the pseudo random generator on the input k_0. So remember, the string k_0k_1 is a string of length 3 bits. So you have 3 bits here, you have 3 bits here, the first 3 bits part I am denoting as k_0, and I call the function G on that input to again obtain a new pseudo random string of length 6 bits, which again, I can divide into 2 parts.

And the right child of my root basically stores the value of the output of the pseudo random generator on the string k_1, which will now give me another pseudo random string of lengths 6 bits which I can parse as 2 chunks of 3 bits, 3 bits each. And then I repeat this process at the first layer of this tree, that means this node will now have the outcome of the pseudo random generator on the 3 bits k_{00} as the input.

And this note will have the output of the pseudo random generator on the seed k_{01}. And again, I obtain an output of 6 bits and so on. So that is the way the internal notes are filled. And similarly I filled the leaf notes also using the same logic to. How exactly I am going to output $F_k(i)$? So imagine, this whole tree is basically the definition of F_k. Now, I have to define what exactly will be the output of this tree on my input i.

So remember, the keyed function F takes 2 inputs, the key input and the actual block input. So with respect to the key input I have defined a tree to be like this. Now I have to define how I take the output of this tree for the input i. So imagine for instance, I want to define or compute the value of this so called function F_k at the input 3. So 3 in binary can be written as 011. And basically, the idea is now I have to just pass this tree based on the binary representation of 3.

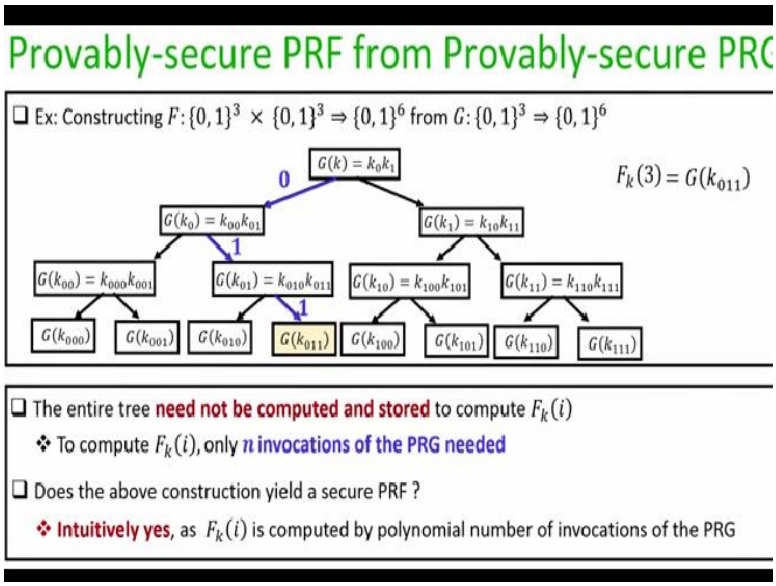
So, I pass the first bit here which is 0, if it is 0, the rule is I go to the left of my current node, so, I start exploring from the root first bit is 0 I go to the left note, the next bit in the binary representation of 3 is 1. So, from my current note, I go to the right and the last bit in the binary

representation of 3s are 1. So, that means from my current note, I go to the right and this will be the value of my function F_k at the input 3.

That is the way I am going to define my function F_k. So, if you see on a very high level basically the way function F_k is defined it is nothing but polynomial number of sequential compositions of the truly random generator. I am basically composing the pseudo random generator G polynomial number of times depending upon the binary representation of my input i. In this case the binary representation was 011.

So, I am basically invoking the function G three times in sequence one after the other where the outcome of the previous invocation of G is serving as the input for the next invocation in a specific way. Depending upon the binary representation of my input i, that is the way you can internally interpret the execution or the construction of this keyed function F_k.

(Refer Slide Time: 10:49)



Now, you might be wondering that whether this construction is efficient or not, because the size of the tree is exponentially large here. It consists of 2^n number of nodes and where n is the security parameter. So that means if I am defining the function F_k like this, then one might feel that both sender and the receiver have to maintain this tree because once they know the value of the k they have to construct the tree like that.

Because they do not know well in advance what is the value of the i that they are going to use it could be end up with any of the leaf nodes. So, had they have the whole tree with them in advance, but storing the whole tree will require them exponential amount of computation. So, intuitively, this construction might look like to be an inefficient construction, but it turns out that the entire tree not be computed and stored to compute the value of discrete function on the input i.

Because depending upon the requirement, that means depending upon the value of i, I can compute the actual part that I need to follow in this tree by just invoking my underlying pseudo random generator n number of times. For instance, if I want to compute the value of the function $F_k(i = 3)$, what I basically need is just 3 invocations of the PRG. That is all. I do not need the remaining invocations of the PRG.

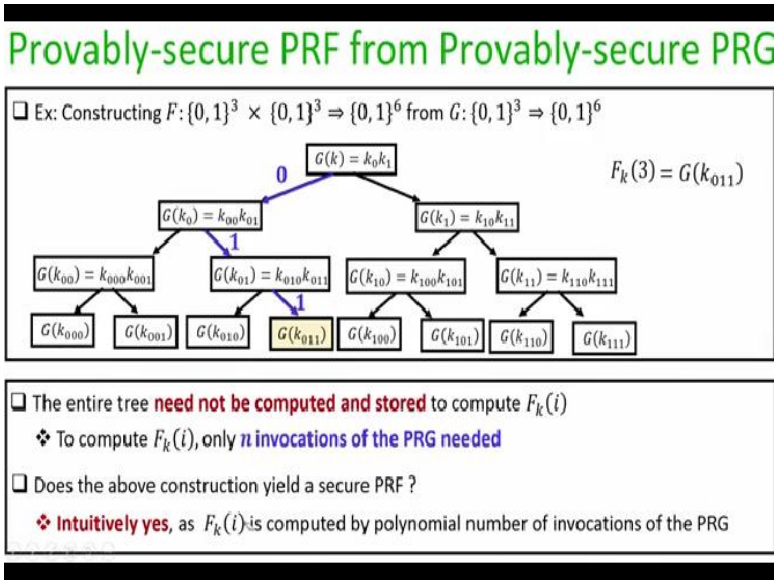
In the same way suppose later I am interested in computing the value of say $F_k(4)$ therefore, in the binary representation is 100, then I do not need the whole tree. I need only this node, namely this invocations of the PRG, followed by this invocation of the PRG, followed by this invocation of the PRG. That means each value of $F_k(i)$ can be just computed by executing polynomial number of instances of the underlying pseudo random generator. That is why this construction is computationally efficient. It does not demand exponential amount of computation.

Now, the big question is this tree construction, or this way of defining the keyed function F_k is indeed going to give me a secure PRG. The answer is yes. Because intuitively what is $F_k(i)$, what is the way I have computed $F_k(i)$. $F_k(i)$ you can interpret as a polynomial number of sequential compositions of PRG.

Remember, when we were discussing pseudo random generator earlier, we had proved rigorously that polynomial number of sequential composition of PRG also gives you a pseudo random generator namely that output will be pseudo random and it will be indistinguishable from the outcome of a corresponding truly random generator. So, in that sense, this way of defining the function F_k based on a complete binary tree is indeed going to define a pseudo random

function. But now, if I want to formalize this intuition into a rigorous proof, then there are a lot of subtleties which are involved here.

(Refer Slide Time: 13:52)

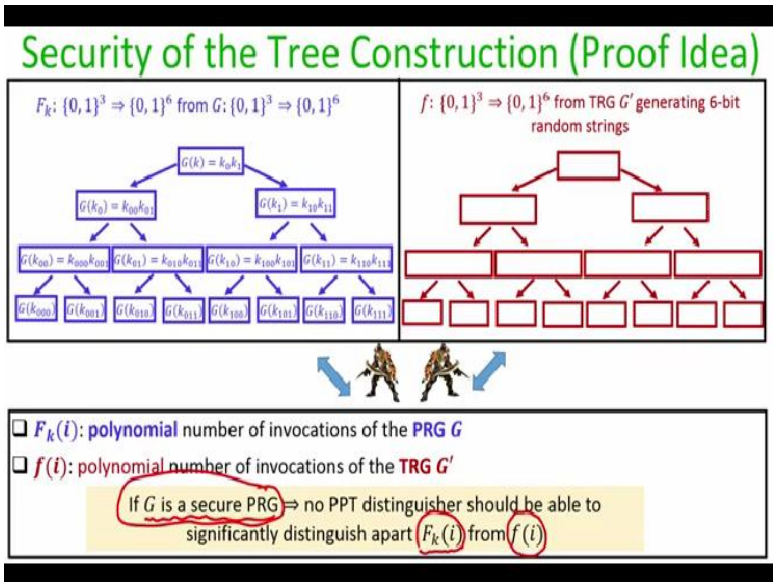


So the actual proof is indeed suttle and it requires lot of advanced technicalities. So, due to the interest of time and since the proof is really out of scope of this course, I will avoid the full formal details of the proof. But if you are really interested to see the complete proof, you can see the proof available in the book by Claude Shannon but let me discuss the overall proof idea.

So, as I said the value of the $F_k(i)$ is nothing but polynomial number of invocations of the pseudo random generator. So, my goal is basically to show that, if an adversary interacts with the keyed function F_k by asking polynomial number of queries where it does not know the value of k, it knows the structure of the tree, but it does not know the value of the k and hence, it does not know what are the pseudo random streams which are stored in the individual nodes.

So, imagine if I have an adversary which is interacting with $F_k(i)$ or a function F_k polynomial number of times, my goal is to show that it should not be able to distinguish the behavior of the tree construction from the behavior of a truly random function, but I cannot directly reduce that indistinguishability to the security of the underlying security of the pseudo random generator, because there are polynomial number of invocations of the pseudo random generator which are involved. So, what basically we required here is the hybrid argument.

(Refer Slide Time: 15:19)



So, let us see the security of the tree construction basically, we have an overview of the proof idea here. And for the demonstration of the proof idea, I take the case where I am constructing a pseudo random function, $F: \{0, 1\}^3 \times \{0, 1\}^3 \Rightarrow \{0, 1\}^6$ this is designed using a pseudo random generator $G: \{0, 1\}^3 \Rightarrow \{0, 1\}^6$.

So, as per the tree construction that we have discussed just now, this is how the function F_k will look like and now, what I am going to do is I am going to compare this tree based construction of the function F_k with an alternate construction, where all the instances of the pseudo random generator G are going to be replaced by a truly random generator G'.

So, what we are basically trying to construct here is we are trying to construct an unkeyed truly random function which takes an input of size 3 bits and it gives you an output of 6 bits. And on a very high level, the construction is exactly the same as the tree construction except that at each node all the invocations of your function G are replaced by G'. So, at root node we just call the function G'.

And since the function G' is a true random generator, it does not take any input. Just gives you some random 6 bit output that will be filled in this root. Then when we go to the left node again,

we invoke the function G', which will give you another 6 bit**,** truly random string. And like that, you can see that each node we are basically just invoking by function G'.

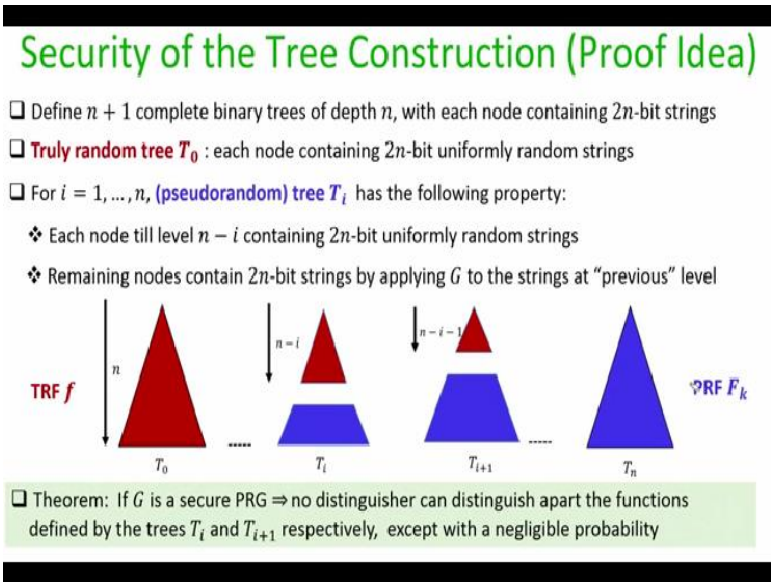
And as a result, each of these nodes in this tree, which is constructed on the right-hand side part will have 2 random values of length 6 bits. So, that is how we have constructed the function f. So, now construction wise difference between the 2 functions that we have constructed is that if we want to the left-hand side k, it defines your function F_k. And if I want to compute the value of this function at some input, i say for instance, if I want to find the value of this function, on your left-hand side on the input is equal to say all 0s.

Then basically I have to follow the path 000 and the value of my $F_k(i)$ will be the value stored here. And we say other hand, if I want to compute the value of the function f that I have constructed in the right hand side on the input all 0s then again I have to follow as the I have to traverse along this tree as per the binary representation of my input i and wherever I stopped the leaf node, the value that is stored there, that will be considered as the value of the function f on the input i.

So, in terms of the way I am computing and obtaining the output of the function, it remains the same in both the functions. What differs in the 2 functions is that the left-hand side tree, all the invocations are for pseudo random generator and the right-hand side tree all the invocations are for true random number generator. Now informally, the proof the idea behind the security of the tree construction that we have given is that if the underlying function G is a secure PRG, then what we are going to show is in the proof that no polynomial time distinguisher should be able to distinguish apart the value of the function, F_k on the input i from the value of the function f on the input i.

That means, it does not matter whether he has interacted with the tree construction on the left hand side polynomial number of time, or whether he has interacted with the tree on the right hand side polynomial number of times, from the viewpoint of the adversary, the interaction should be almost identical except with negligible probability. If indeed my function G is a secure PRG. So that is what is basically the overall idea of the proof.

(Refer Slide Time: 19:39)



That is what I have to show. And the idea behind a proof here is we basically define n + 1 complete binary tree, each of depth n, where each node is going to store 2n bit strings, but in a different way. So, let us start with the tree T_0, which is actually a tree of depth in complete binary tree of depth n where each of the nodes basically consist of a uniformly random 1 n-bit string.

And this is nothing but the way a truly random function f will behave as per the tree construction and my ith tree i will be as follows. In my i^{th} tree T_i, the first n - i levels, all the nodes in those n - i levels will consist of true l-bit uniformly random strings, whereas all the remaining levels will consist of pseudo random strings by applying the key mechanism or the key construction to the node at the previous level.

If I go to the i = 1th key the way it differs from the ith key is that it will have one layer less of pseudo random strings and one layer more of pseudo random strings compared to the previous string. That means in the $i + 1^{th}$ tree the first n - i – 1 layers of node will consist of uniformly random strings of length 2n-bits. And the remaining layers of node will consist of pseudo random strings of 2n- bits by applying the pseudo random generator G on the previous level and so on.

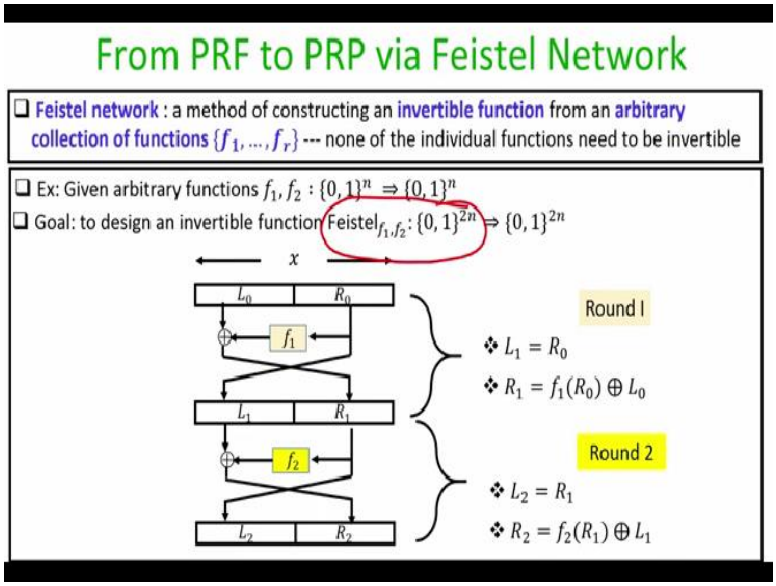
And like this, if I continue my n^{th} , $n+1^{th}$ tree, T_n basically is the way I have defined the function F_k, that means all the nodes consist of pseudo random strings of length 2n-bits by applying the pseudo random generator G to the value of the 3 nodes at the previous level. So, that is the way I have defined n + 1 trees. And each of these trees basically defines a construction of a function mapping n bit strings 2 n bit outputs.

So the first tree basically defines the way a truly random function will operate. And the last 2 defines the way we have constructed the keyed function F_k and the overall idea behind a security proof of the keyed construction is that we can prove that formally, if my underlying G is a secure PRG, then the behavior of the function defined by the tree T_i and the behavior of the function defined by the tree T_{i+1} are computationally indistinguishable from the viewpoint of an attacker, who makes polynomial number of queries to the function defined by the tree T_i, or make polynomial number of queries to the tree defined by for to the function defined by the tree T_{i+1}.

And this argument we can reduce by giving a reduction-based argument. And we can show that if at all, there is an adversary who can distinguish apart the behavior of the function F_i from the function F_{i+1}, then it knows how to distinguish apart the behavior of a truly random generator from a pseudo random generator, that is the overall idea.

Since there are polynomial number of intermediate hybrids, in between my truly random function F, and my function F_k which I have defined, I can say that the overall the probability with which an adversary can distinguish apart the behavior of the function f from the behavior of the function F_k is the summation of polynomial number of negligible quantities, which is again on a negligible probability. That is overall idea of the security proof, but the actual formal details are really involved in suttle. And that is why I due to the interest of the time I am skipping.

(Refer Slide Time: 23:27)



So, now we will see that if we are given a pseudo random function, how we go about to construct a pseudo random permutation by a very interesting primitive, which we call us Feistel network, and this is a very powerful cryptographic primitive or construction, which we again encounter when we will see the practical instantiations of pseudo random permutations namely when we will discuss about the construction of the DES.

So the basic idea here behind a Feistel network is that it gives you a method of converting invertible function from arbitrary collection of several functions which need not be invertible. So what exactly that means. So for demonstration purpose assume you are given 2 arbitrary functions $f_1,\ f_2 : \{0,\ 1\}^n \Rightarrow \{0,\ 1\}^n$, which may not be invertible. That is why I am saying they could be any arbitrary functions.

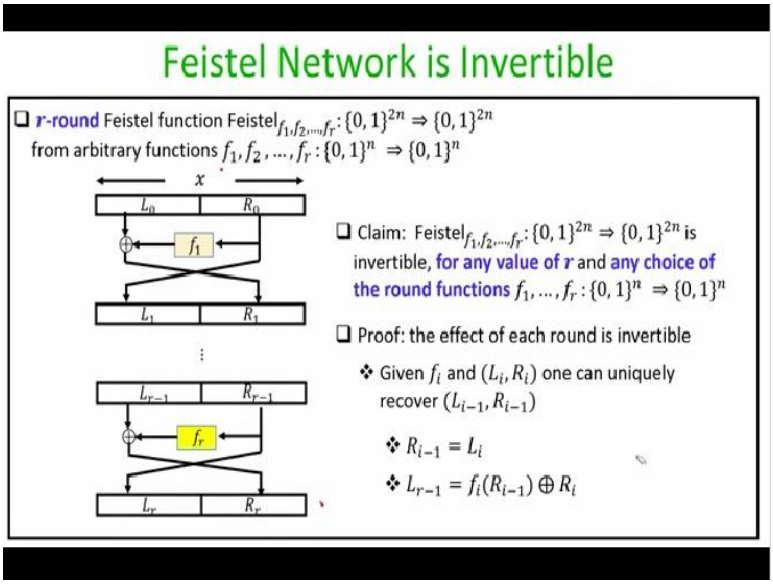
My goal is to use $f_1,\ f_2 : \{0,\ 1\}^n \Rightarrow \{0,\ 1\}^n$ and define a new function $\text{Feistel}_{f_1,\ f_2} : \{0,\ 1\}^{2n} \Rightarrow \{0,\ 1\}^{2n}$ such that resultant function is invertible. So that resultant function I call denote as $\text{Feistel}_{f1,f2}$, because I am composing the 2 arbitrary functions f_1 and f_2 in a specific way, which we will see soon to obtain an invertible function.

So here is how the composed function $Feistel_{f1,f2}$, will work or look like. So, it will take an input of size 2n-bits and it has to produce an output of size 2n-bits by somehow composing the functions f_1 and f_2. So, what we are going to do here is we will pass the input x as 2 chunks of n-bits, n-bits each and the overall construction will be interpreted as a sequence of 2 rounds. So, in round 1, I am going to construct or convert this input x into an intermediate output, which again I will parse as 2 parts which I call as the left half and the right half.

The way this $L_1 \| R_1$ is computed from $L_0 \| R_0$ is as follows: the $L_1 = R_0$ part and the $R_1 = f_1(R_0) \oplus L_0$. So, since my function f_1 takes an n-bit input, and my R_0 part is also n-bit input, that is fine. And output of the function f_1 is again n-bits, which can be exored with an L_0 part, which is of n-bits to give me an output R_1, which is of n-bits. So that is the rule or that is the way I am going to use my function f_1 for the first one.

Now, once I obtained the intermediate output denoted as of concatenation of L_1 part and R_1 part I do the same principle, but now in the second round, I am going to use the second function. And that is why this is a 2-round construction. In the second round again, my $R_1 = L_2$ and my $R_2 = f_2(R_1) \oplus L_1$. That is the way the function $Feistel_{f1,f2}$ will look like.

(Refer Slide Time: 27:00)



So, in general, if you are given r arbitrary functions, $f_1, f_2, \ldots, f_r : \{0,1\}^n \Rightarrow \{0,1\}^n$, then I can compose that by applying this logic which we had seen in the previous example, sequentially over time. And what I obtain is function which I denote as $\text{Feistel}_{f_1, f_2, \ldots, f_r} : \{0,1\}^{2n} \Rightarrow \{0,1\}^{2n}$. So, the idea behind is that I apply the same logic that we had seen in the last example r times. If we are in the ith round, I apply the ith round function namely f_i.

Now, you might be wondering whether the resultant function Feistel composed which consists of basically a sequence of r compose r functions is indeed going to give you an invertible function or not. So, I claim that it does not matter what exactly is the choice of your underlying functions $f_1, f_2 \ldots, f_r$. That means it does not matter what exactly are your own functions. So, I call this individual functions which I am applying in the individual round or the round function, the claim that we are going to make is, it does not matter what exactly is your round functions, they may not be invertible it could be any arbitrary functions. The way we are composing this r individual functions that result in function is always going to give you invertible functions, irrespective of how many times you do it.

The idea behind the proof is the effect of every round can be unique. The idea behind the proof of this claim is that the effect of each round is invertible irrespective of how what exactly is your function. So for instance, let us see that whether we can reverse back that effect of rth round function, that means, imagine you are given the output of this Feistel function namely you are given L_r concatenated with R_r and the question is can I uniquely go back to the previous intermediate output given that I know the rth round function.

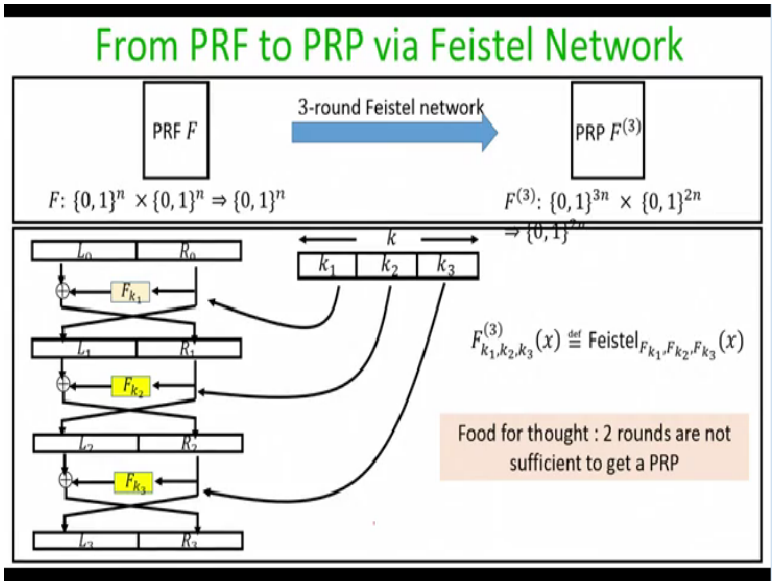
It turns out we can do that by the description or by the nature the way we have done the composition we know $R_{i-1} = L_i$. So, I can always go back from the current left half to the previous right half by this route. And I know that $L_{r-1} = f_i(R_{i-1}) \oplus R_i$

.

That is the way I can uniquely go back from my current state to the previous state. And then I can repeat this argument and go back one level up. Again, I can repeat this argument and again can go back one level up and all the way I can go back to the input L_0 and R_0. That means it does not matter what exactly is that type of the individual round functions, the way we have actually compose these functions.

Given the final outcome of the composed function, I can always uniquely go back to the actual input. And in that sense, this function, the composed function is an invertible function.

(Refer Slide Time: 30:51)



So now let us see how we go about constructing pseudo random permutation given that we have provably secure pseudo random function. So, I am assuming that I have a construction of a provably secure pseudo random function for simplicity, I assume that a key land, block land and output land are all n-bit strings. And I am going to use a 3-round Feistel network that means, I will be now applying 3-round functions.

I will end up obtaining a keyed permutation where the length of the key will be 3n-bit strings and the block length will be 2n-bits and output will be 2n-bits and keyed function, which I denote as $F^{(3)}$ can be proved to be a keyed permutation. So, since the key is going to be of length 3n-bits, I can interpret it as 3 chunks of or 3 independent chunks of n-bits, n-bits, n-bits.

I am going to apply the Feistel network 3 times or basically I am going to apply the 3-round Feistel network which basically means I have to compose 3-round functions. Now, basically in each of rounds, I am going to invoke the underlying pseudo random function with independent keys. So, the way I am going to define my keyed permutation is nothing but the composed Feistel network, where the first-round function is the keyed pseudo random function on the first n-bits of the key of my pseudo random permutation.

The second-round function is going to be F_{k2} namely, my invocation of pseudo random function with k_2 part of the key. And the third-round function will be F_{k3} namely the invocation of the pseudo random function with the last n-bits of the key. That is the way I am going to compose the pseudo random function 3-times using the structure of the Feistel network.

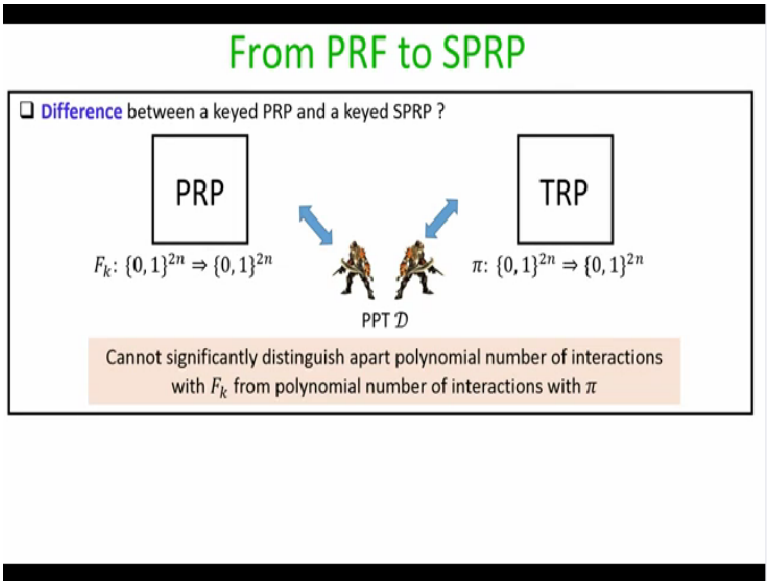
So that is the way I have defined. So, my first-round function is F_{k1}. And by applying F_{k1} as my first-round function, I go from L_0R_0 to L_1R_1. Given L_1R_1, I apply F_{k2} or treat F_{k2} as my second round from and go from L_1R_1 to L_2R_2. And finally, by applying F_{k3} on the input L_2R_2, I obtained L_3R_3. And that is what will be the outcome of the function $F^{(3)}$ under the key k_1, k_2, k_3 on the input, $L_0 \| R_0$.

That is the way I am going to define a keyed permutation. So, it is easy to see that a resultant composed Feistel function is indeed an invertible function we had already proved that. What is left is to show that why this magical 3-round construction is going to give me a pseudo random permutation? That is why this keyed permutation is indistinguishable from the behavior of a truly random permutation.

Well, again, the proof is slightly involved, and I leave the complete details due to the lack of the time. You are referred to the book by Claude Shannon for the actual proof but you have to believe me that if I compose this pseudo random function 3-times with independent keys as per the structure of the Feistel network, then the result and constructions is indeed a keyed permutation. Now, an important point here is, why not 2-rounds? why we have to compose this Feistel network 3 times why we require 3 why not 2 rounds?

It turns out that if I use a key of size only say 2n-bits, and I apply only 2-round functions, and the resultant keyed permutation is not pseudo random, it is easily distinguishable from a corresponding truly random permutation. And that is why it is only when we compose 3-times we actually get a pseudo random permutation. So this is left as an assignment for you. You have to now think that why exactly 2 rounds are not sufficient why 3 rounds.

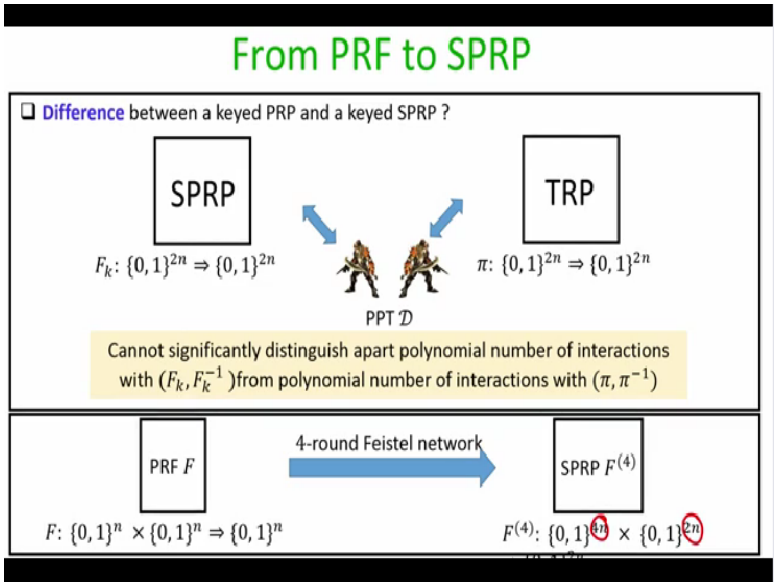
(Refer Slide Time: 34:53)



Now let us finally see that how do we go about constructing strong pseudo random permutations from a pseudo random function. Before that let us first see what exactly is the difference between a pseudo random permutation or keyed pseudo random permutation and a keyed strong pseudo random permutation?

So, if I consider pseudo random permutation, which is a keyed permutation mapping say, 2n-bit strings to 2 n bit strings, when I say it is a pseudo random permutation, then it means that no polynomial time distinguisher can distinguish apart the interaction with this keyed permutation from an unkeyed truly random competition.

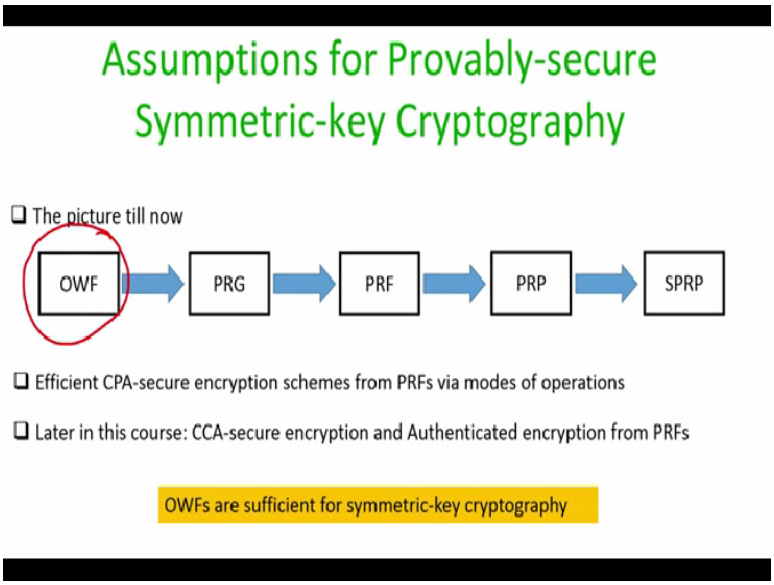
(Refer Slide Time: 35:29)



But if I say that I have strong pseudo random permutation, then it is a special type of keyed permutation, which should be indistinguishable from a corresponding truly random permutation, even if my distinguisher gets access to not only the outputs of the permutation, but also to the inverse of the permutation. That means, the keyed permutation should be indistinguishable even if the adversary is getting interaction or oracle access to the function F_k as well as the inverse of the function.

So, imagine we are given a provably secure pseudo random function, which we now have to construct using that reconstruction. It turns out that if we do a 4-round Feistel network, namely if we use 4-round functions and compose it as per the structure of the Feistel network, then we end up getting a keyed strong pseudo random permutation mapping 4n-bit strings and blocks of size 2n-bits to an output of size 2n-bits. And again, the proof is slightly involved, which I am leaving due to the interest of the time you are referred to the book by Claude Shannon.

(Refer Slide Time: 36:42)



So, that is the overall idea here. So now if we see, if you look into the assumptions that we required for provably secure symmetric cryptography. The picture till now is as follows: we know that if you are given one-way functions then using the Goldreich-Levin theorem and hard-core predicate we get provably secure pseudo random generator. And in this lecture, we had seen that from pseudo random generator we can construct provably secure pseudo random function using which we can construct provably secure pseudo random permutation.

And then it can be further used to construct provably secure strong pseudo random permutation. And we also know that how we can construct efficient CPA secure encryption scheme from PRFs by using modes of operation. Later in this course, we are going to see that how we can in fact construct more powerful symmetric encryption process namely authenticated encryption and CCA secure encryption just using pseudo random functions.

So it turns out that everything just depends upon the existence of one way function that means if you want provably secure constructions of provably secure CPA secure encryption scheme, provably secure CCA scheme, provably secure authenticated encryption scheme, then it is suffice to just have one way function. That means it is enough you have just one-way function you can get everything for free.

And later on in this course, when we will discuss public key cryptography, we will see that how exactly we can go about and construct one way functions based on specific number theoretic hardness assumptions. So, everything boils down to the existence of one-way functions.

So, that brings me to the end of this lecture. Just to summarize in this lecture, we had seen very high-level overview of how we give provably secure constructions of pseudo random function, pseudo random permutation and strong pseudo random competition. Thank you!