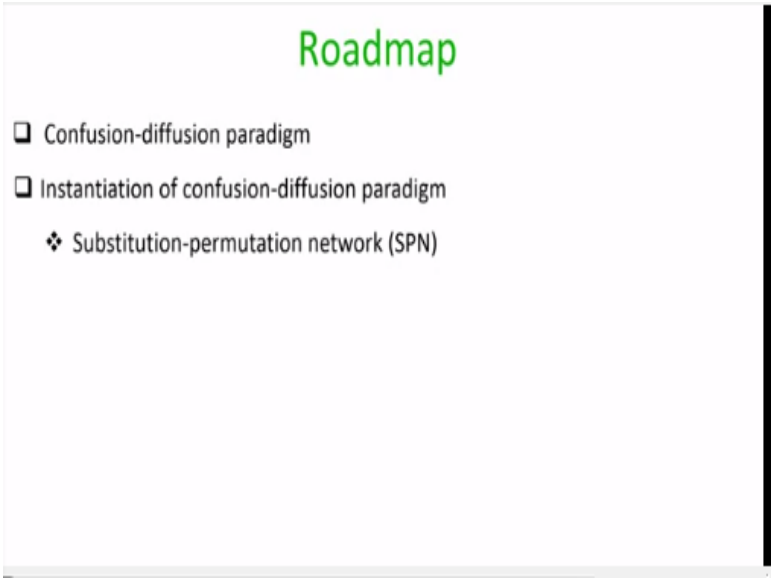
Foundations of Cryptography
Prof. Dr. Ashish Choudhury
(Former) Infosys Foundation Career Development Chair Professor
International Institute of Information Technology-Bengaluru

Lecture-19
Practical Constructions of Block Ciphers Part I

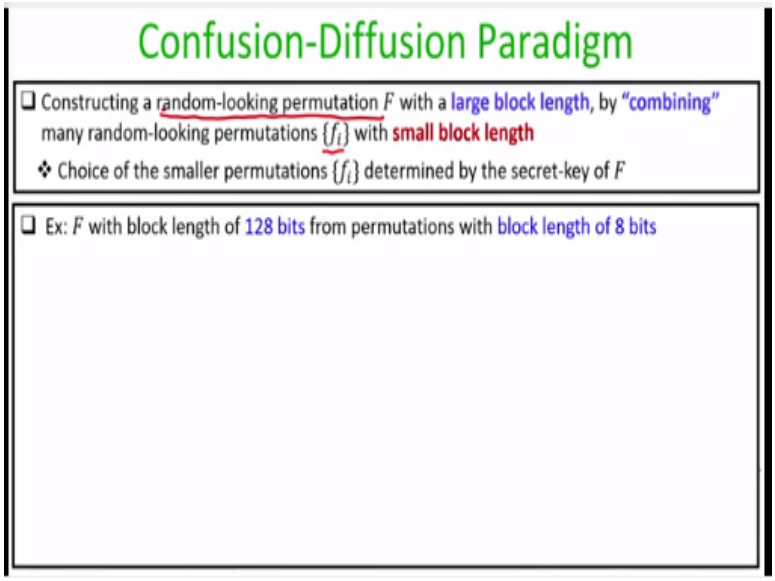
Welcome to lecture 18, in this lecture we will discuss about the practical constructions of block ciphers. And this will be the part 1 of the discussion on the practical constructions of block ciphers.

(Refer Slide Time: 00:41)



So, the road map of this lecture is as follows, we will start discussing about confusion diffusion paradigm and then we will see how to instantiate the confusion diffusion paradigm using substitution permutation network or SPN. Later on in during our part 2 of the discussion on practical constructions block cipher, we will see that how SPN plays a very crucial role in the design of one of the most popularly used block ciphers, namely DES.

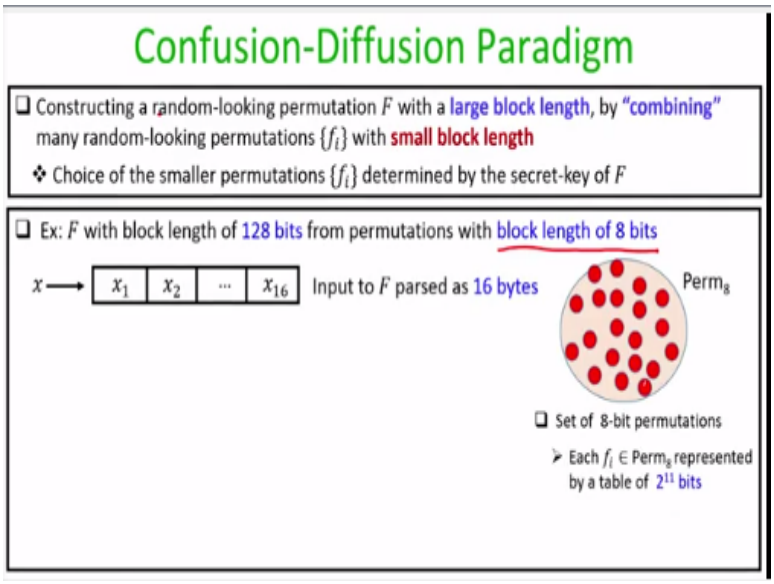
(Refer Slide Time: 01:10)



So let us start our discussion on confusion diffusion paradigm. So, this paradigm was introduced by Claude Shannon. And the underlying idea here is that we are interested to design a random looking keyed permutation, right. So that is our goal, we want to construct a random looking keyed permutation F with a larger block length by combining several random looking permutations, which are also keyed permutations.

But with a small block length, so the random permutations with small block length those are denoted by f_i. And for basically the confusion diffusion paradigm does is it tells you how to compose or to combine this random looking permutations with small block length to obtain a random looking permutation with a larger block length, right. And the way we do the composing is in such a way that the choice of the smaller permutations f_i are determined by the secret key of your larger sized permutation.

(Refer Slide Time: 02:17)

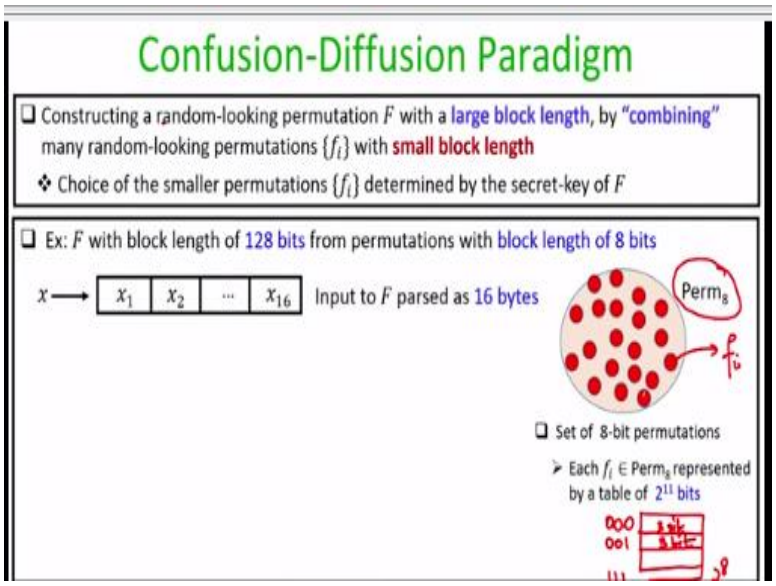


So, to demonstrate how exactly the confusion diffusion paradigm works, let me take an example where my goal is to construct the keyed permutation F with a block length of 128 bits using several keyed permutations of block length of 8 bits, right. So, I beg your pardon here, so the smaller size permutations, they are not keyed permutations, they are unkeyed permutations. There will be truly random permutations and what basically the confusion diffusion paradigm does is:

It tells you how to compose these small block size unkeyed permutations and obtain a larger blocks length, the keyed permutation. So for the demonstration purpose, we will assume that we have several random permutations, mapping 8 bits to 8 bit strings. And using that our goal is to compute or construct a keyed permutation F of block length of 128 bits. So the way we are going to design the function F is as follows.

So we parse the input f_x for the function F as a sequence of 16 bytes, so those bytes I am denoting as x_1, x_2, x_{16}. And the reason I am splitting it into bytes is because we are given at our disposal several permutations, mapping 8 bits to 8 bits, right. So that is why the input x for the keyed permutation F is divided into bytes here.

(Refer Slide Time: 03:44)

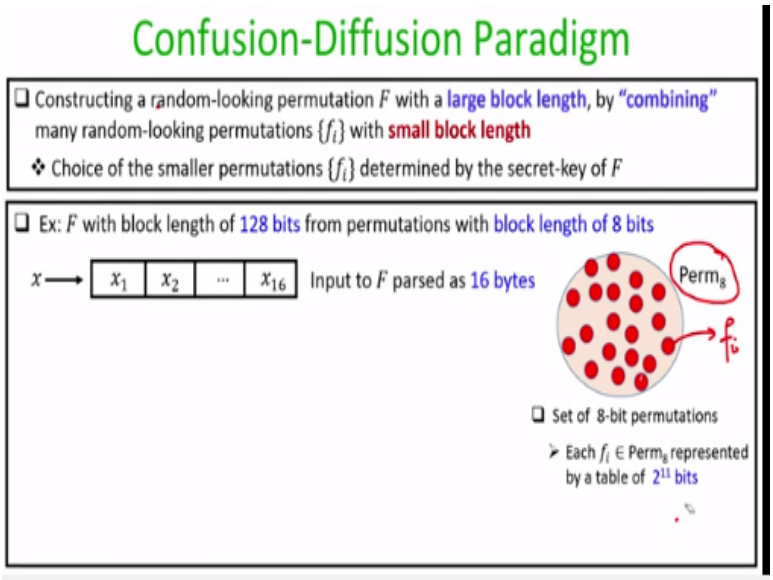


So imagine that you have the set of all permutations mapping 8 bit string to 8 bit strings. So that set, I am denoting by this notation from sub 8 right. So each of this small circles, you can imagine that it is a permutation mapping 8 bit string to 8 bit string right. And each of these permutations, say, if I take this particular permutation f_i. You can interpret it as a table consisting of 2^8 entries, right.

So you can imagine it as a table consisting of 2^8 entries where the first entry denotes the value of the permutation on the input all 0. The second entry denote the value of the permutation on the input 001 and like that the last entry denote the value of the permutation on the input 111. So, like that you will have 2^8 rows, and each row will basically consisting of 3 bits, and 3 bits, namely the value of permutation on the corresponding inputs.

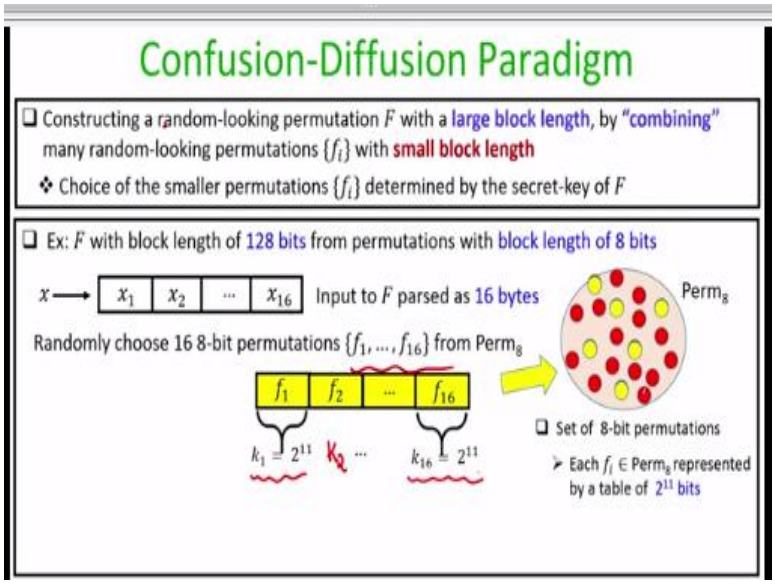
So, in that sense, I can imagine or interpret each of these small permutations f_i as a table consisting of 2^{11} bits, right.

(Refer Slide Time: 04:57)



So, like that you have several such tables and each such table I am denoting as a red circle here. And a collection of all those red circles basically is my set Perm_8 ok.

(Refer Slide Time: 05:07)

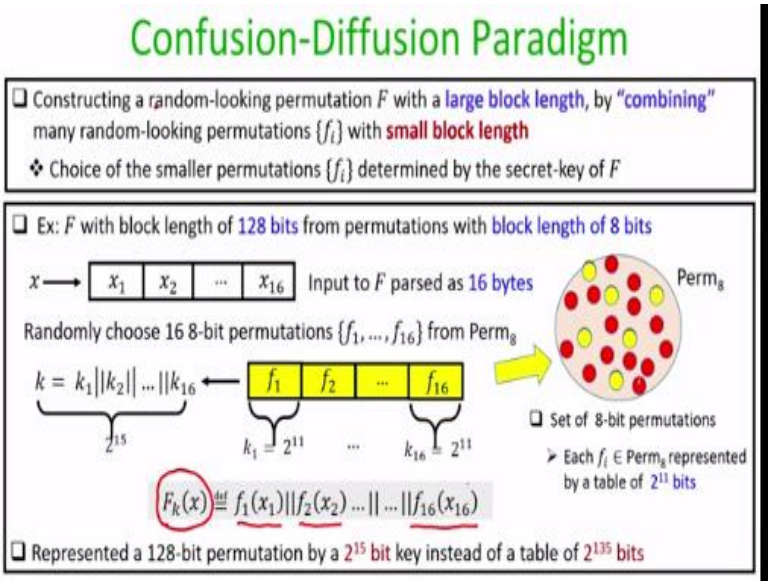


So, the way we are going to construct the keyed permutation F is as follows, we randomly choose 16 permutations f_1, f_2, f_{16} from this bigger set. And say the randomly chosen permutations of 16 permutations that I have chosen are denoted by this yellow circles. And each of these 16 permutations mapping 8 bit to 8 bit strings can be individually interpreted as a string of 2^{11} bits.

This is because as I said earlier, each of the permutations can be interpreted as a table consisting of 2^{11} bits. So, the choice of the first small permutation that corresponds to a string k_1, the choice

of the next permutation that will be the string k_2. And like that the choice of the 16th permutation mapping 8 bit string to 8 bit string can be interpreted as another string of 2^{11} bits ok.

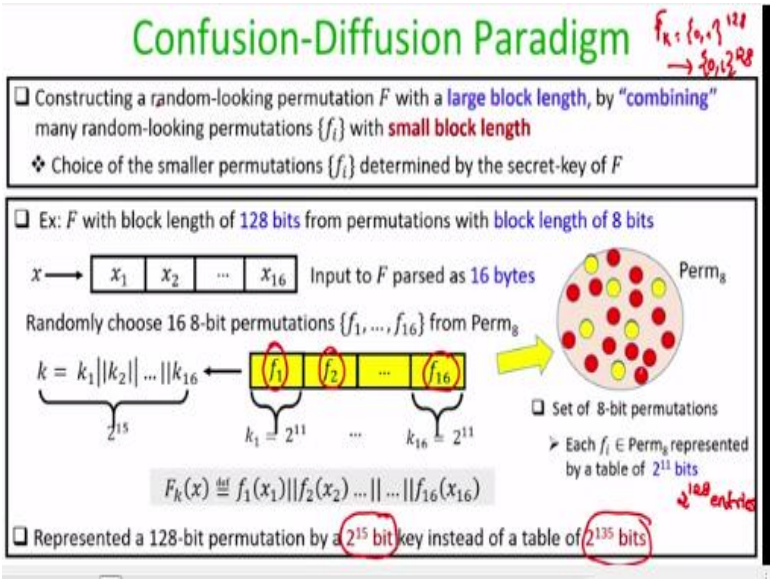
(Refer Slide Time: 06:09)



And now once we have randomly chosen the 16 permutations, which are going to be used for the construction of the function F, the overall key for the big permutation F is the concatenation of the individual tables. Namely, the concatenation of k_1, k_2, k_{16} and if I concatenate all the strings, I obtain the overall key for the function F. So, it is easy to see that the key size for the function F which we are interested to construct will be of 2^{15} bits.

Because each of the smaller permutations which I am going to use is denoted by 2^{11} bits and I have 16 such permutations. So overall the key size for the bigger permutation F is 2^{15} bits. And once I have decided the value of the key, the output of the keyed function F_k on the input x is determined to be the value of the first small size permutation f_1 operated on the first byte, concatenated with the value of the second smaller permutation f_2 on the second byte of the input x. And like that the value of the 16th smaller sized permutation on the 16th byte of the input. And if I concatenate all these individual outputs, that is what is going to be my output of the keyed function f_k on the input x. So that is the way we are going to construct a function F using several smaller size permutations. Now the advantage of constructing the bigger function F_k like this is as follows.

(Refer Slide Time: 07:56)

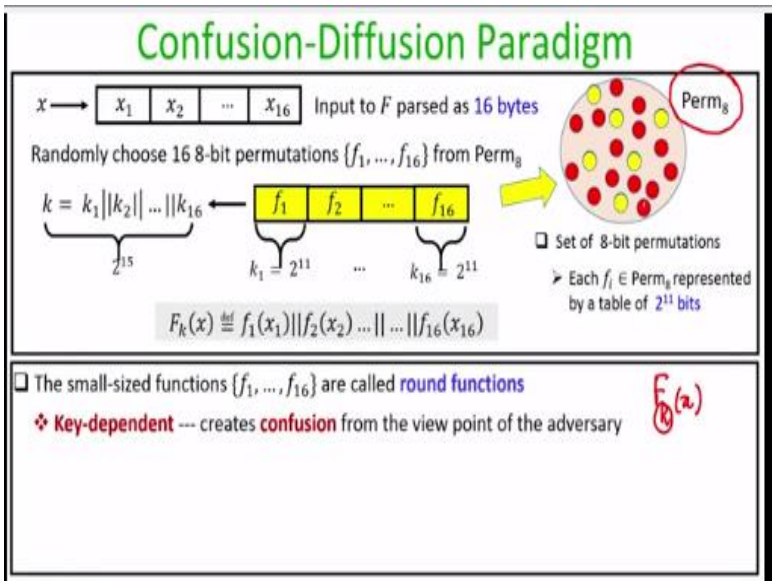


If I would have directly tried to construct a function F_k, right mapping say 128 bit strings to 128 bit strings. Then that would have required me to actually store a table consisting of 2^{135} bits, because that table will have 2^{128} entries. And each entry would have further consisted of 8 bits. So that is why the overall size of that table would have been 2^{135} bits.

But the way we have constructed the function F_k by combining several 16 smaller permutations, I just need to store a key of size 2^{15} bits. And it is enough if I just store the value of the k namely, the description of the 16 smaller permutations which I have chosen. And that suffices for evaluating the value of F on any input x. So that is the advantage of constructing the keyed function or keyed permutation F_k by this confusion diffusion paradigm.

So now you might be wondering why exactly the name confusion and diffusion in this term confusion diffusion paradigm, what exactly causes confusion and what exactly causes diffusion.

(Refer Slide Time: 09:13)

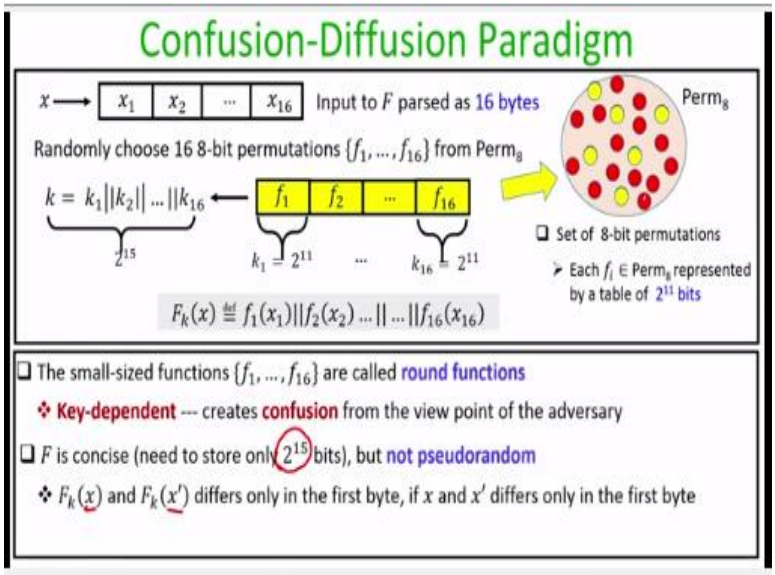


So let us go into the details, so this is the way in our example we have constructed the keyed function F_k. So, the smaller size function f_1, f_2, f_{16} which we have chosen here, they are called as round functions and they are dependent on the key, right. So that creates a confusion from the viewpoint of an adversary. In the sense if an adversary wants to compute the value of $F_k(x)$, but he is not aware of the value of k.

Then the adversary does not know what exactly are the 16 round functions I am going to use. And from the viewpoint of the adversary, it could be any 16 functions from this collection of all permutations mapping 8 bit strings to 8 bit strings. And a size of this collection of all permutations mapping 8 bit strings to 8 bit strings is enormously large. So that means adversary will be completely confused or clueless what exactly is the value of the key, that means what exactly are the 16 round functions.

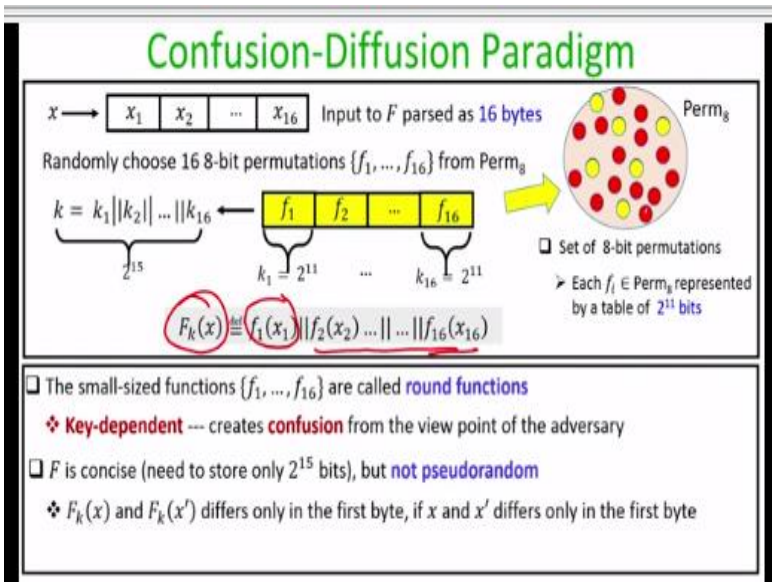
And hence, adversary cannot predict what exactly is going to be the value of $F_k(x)$ even if it has seen several value of $F_k(x)$, for several x of it is choice in the past. So, that creates the confusion aspect in this paradigm.

(Refer Slide Time: 10:30)



However, it turns out that even though the description of the keyed function F_k which we have constructed is very concise. Namely, it requires us to store only 2^{15} bits, it is not pseudo random, right. So recall that the property of the pseudo random permutation is that. If I have 2 inputs x and x' and if I compute the value of $F_k(x)$ and $F_k(x')$ with respect to the same key, then even if x and x' differs in a single bit or a single byte, the outputs should be significantly different. That is what we expect from a truly random permutation and if I say that F_k is a pseudo random permutation, then basically I expect almost similar behavior from the resultant $F_k(x)$ function.

(Refer Slide Time: 11:26)

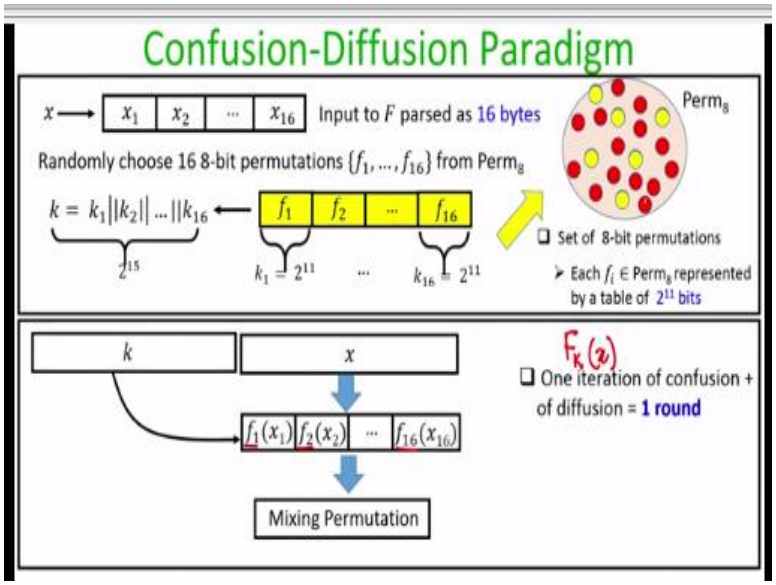


But it turns out that the way we have constructed the function $F_k(x)$, this property is not achieved. It is easy to see that if I have 2 inputs, x and x' which differs only in the first byte, then

the output of $F_k(x)$ and $F_k(x')$ will differ only in the first byte. All the remaining 16, 17, 15 bytes of the output of $F_k(x)$ and $F_k(x')$ will be exactly the same. And that is not what exactly you expect from a pseudo random permutation.

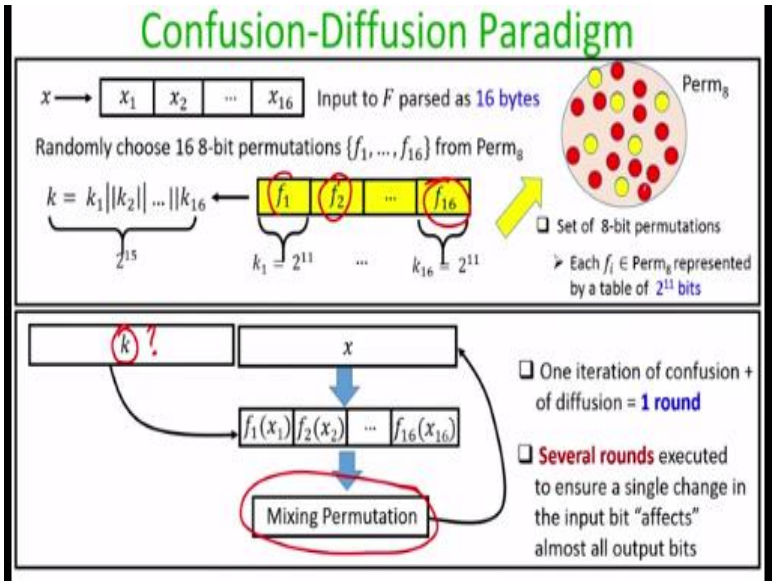
So the way we get around with this difficulty is that we actually apply the logic that we have used in the construction of $F_k(x)$ several number of times by introducing what we call as a diffusion step.

(Refer Slide Time: 12:05)



So let me go into a little bit more detail, so imagine we have the input x for the function F_k, which we are interested to construct, and we have the value of the key. What we do is that depending upon the value of key, we determine the smaller sized permutations which we are going to apply on the individual bytes of x. And after we obtain the outcome of the individual round functions on the respective bytes, what we do is we apply a mixing permutations. Namely, we just shuffle the bits of the intermediate outcome that we have just obtained.

(Refer Slide Time: 12:44)



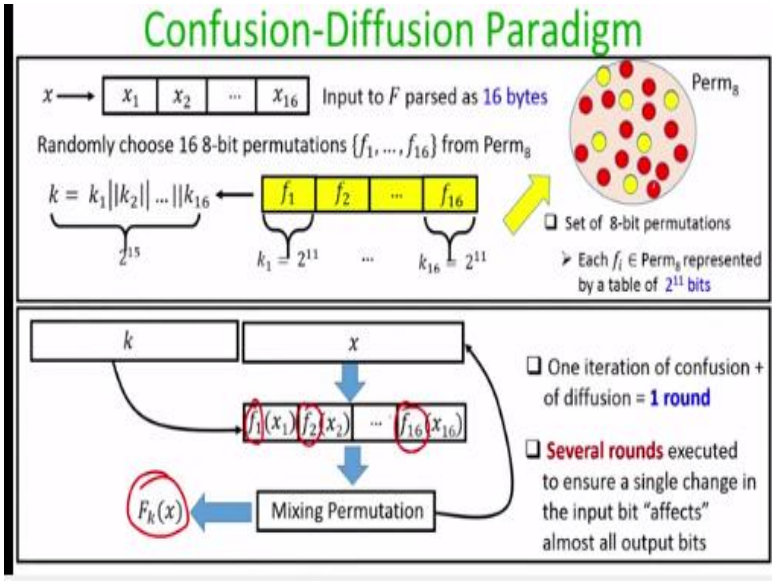
And what we say is that this one iteration of confusion followed by one iteration of this mixing permutation constitutes 1 round. And then we repeat this process again, that means we just do not output whatever we obtain after 1 round as the output of $F_k(x)$, whatever output we obtain at the end of first round, that is considered to be the x input for the next iteration. And again, depending upon the value of the key, we determine the round functions that we are going to use.

We apply the individual round functions on the intermediate output and again we do a mixing permutation. And then we repeat this process for several iterations for several fixed number of rounds. And by repeating this process for several number of rounds, it is ensured that even if there is a single change in the input bit, it affects over several output bits. Because every iteration, the result of mixing permutation will cause the bits of the intermediate output to shuffle around.

And that is what creates a diffusion from the viewpoint of the adversary and that is why the name confusion diffusion paradigm. The confusion is created because the value of key is not known for the adversary. And that is why the choice of the round functions would not be known to the adversary. And diffusion is caused because of the mixing permutation which ensures that after every at the end of every round whatever output we obtain that is shuffled around.

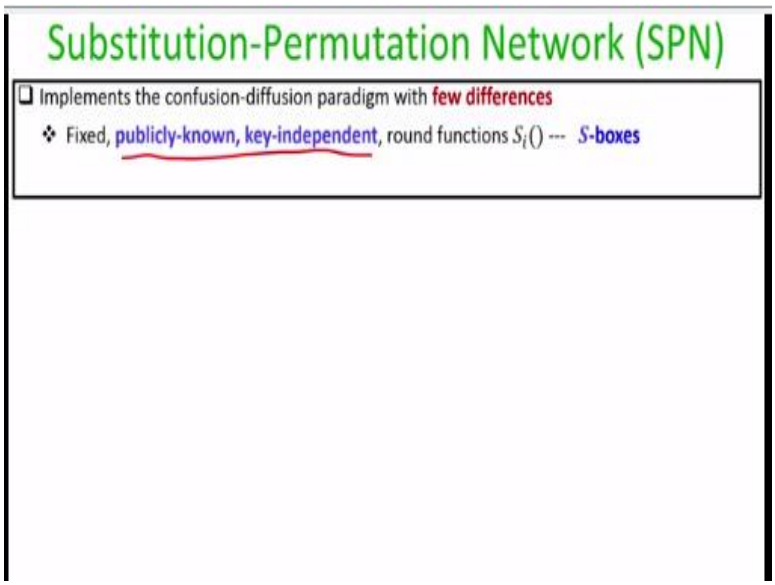
And that is as a result when we go to the beginning of the next iterations, the next round functions will be applied on different bytes. So that creates a diffusion, which ensures that even if there is a single change in the input bit, that affects several output bits and ensures that the overall keyed permutation $F_k(x)$ that we obtained behaves like a pseudo random permutation, right.

(Refer Slide Time: 14:39)



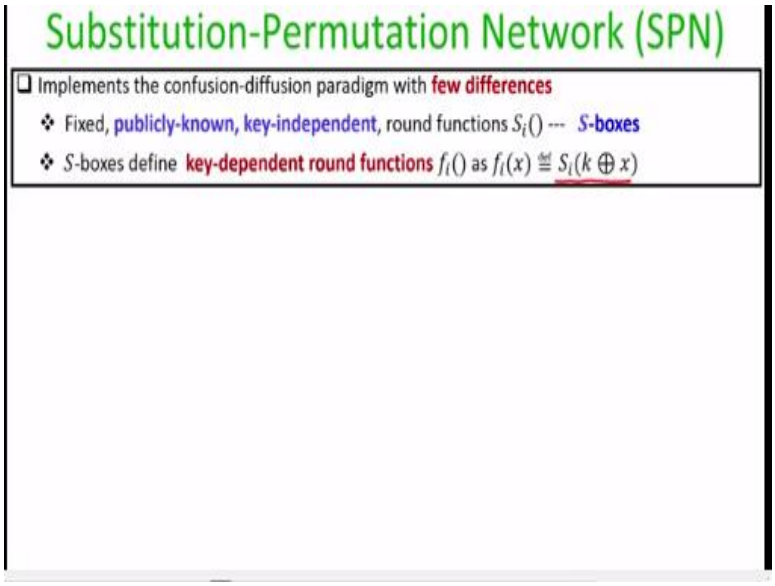
So that is on a very high level, the confusion diffusion paradigm. And after we do this process for several number of iterations, the resultant output is denoted as $F_k(x)$.

(Refer Slide Time: 14:49)



So now let us discuss about substitution permutation network or SPN in short, which implements the confusion diffusion paradigm. And this SPN is a very important building block used in the construction of practical block ciphers like DES, which we will discuss in the next lecture. And this SPN implements the confusion diffusion paradigm, but with few differences. The most important difference is that instead of choosing round functions which are key dependent, we will be now using key independent and publicly known round functions, which I denote by S_i or which are also called us S boxes. Because they are known as substitution boxes that means, if I go back and if I see the description of the confusion diffusion paradigm. In the confusion diffusion paradigm, the confusion was coming because the value of key was determining the round functions that we are using.

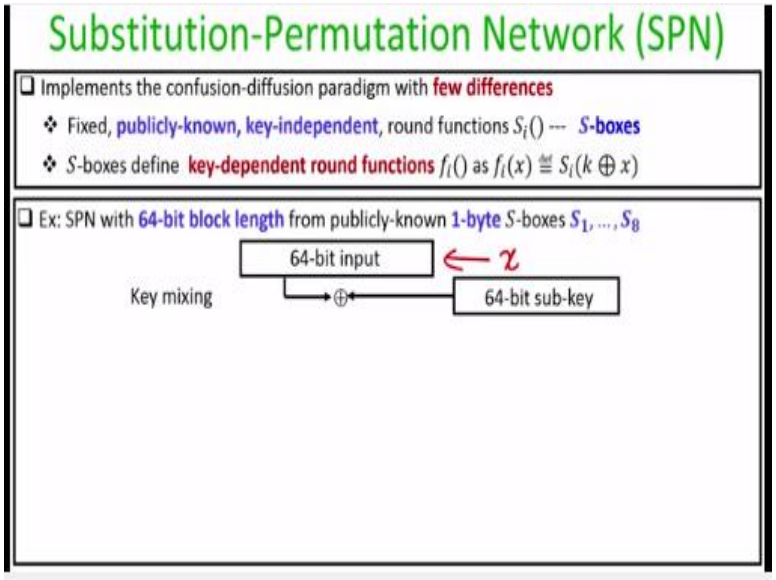
(Refer Slide Time: 15:51)



But in the substitution permutation network, there are no round functions which are key dependent, everything is publicly known. So, now you might be wondering that if the round functions that we are going to use in SPN are publicly known and key independent, how exactly the confusion is going to be brought from with respect to the viewpoint of the adversary. Well, the confusion comes because even though the S boxes are publicly known, the S boxes are applied on an input, which depends upon the value of the key. Namely, if I want to apply the S box on an input x, then we do not directly compute the value of the S box on the input x. But rather on the value of x, XORed with the value of the key and that ensures that even though the value of the description of the S box is publicly known. The exact input on which the S box is

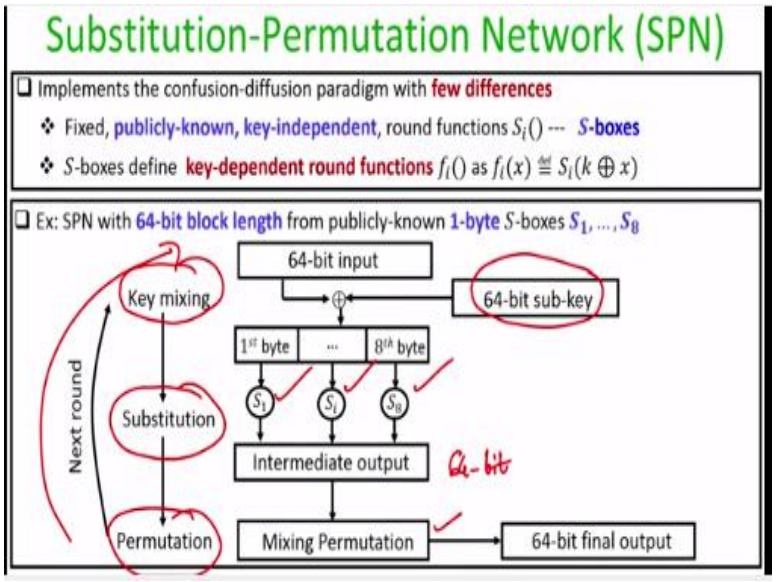
operated upon is not known from the viewpoint of the attacker. And that is what creates that confusion aspect of the confusion diffusion paradigm.

(Refer Slide Time: 16:52)



So, again to demonstrate how exactly an SPN works, let us discuss how exactly we are going to construct a keyed permutation with a block size of 64 bit. Assuming that we are given 8 S boxes, right. So those as boxes we denote as S_1, S_2, S_8 and they are publicly known that they are key independent. So this is the x input for my keyed permutation that I am interested to construct which I denote as x. And the first step in the SPN architecture will be the key mixing step, where we use a 64 bit sub key and mask it with my current x input.

(Refer Slide Time: 17:36)



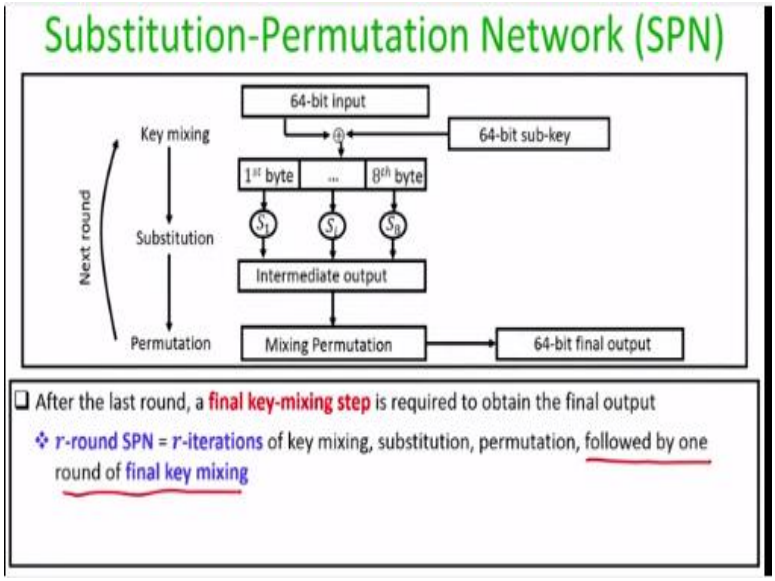
And whatever output I obtain that I parse it as collection of 8 bytes, and now this individual 8 bytes go through the 8 S boxes, which are publicly known. So the first step where actually we do the masking of the input : current input with the sub key, that step is denoted as the key mixing step. And this key mixing step is followed by a substitution step, where the outcome of the key mixing step are interpreted as individual bytes and they go through the individual S boxes.

And as a result, now I obtain a 64 bit intermediate output and the substitution step is followed by a permutation step, where we do the shuffling. Namely, we apply a mixing permutation and this mixing permutation also will be publicly known. So, in this whole architecture, everything will be publicly known except the value of the key, the adversary would not be knowing the value of the key.

Other than that, it will be knowing the entire architecture, it will be knowing the description of the S boxes, it will be knowing the description of the mixing permutation and so on. And once we apply the mixing permutation that finishes 1 round, and then we apply this process again. That means we go to the next round that means whatever intermediate output we obtain at the end of the mixing permutation that serves as the x input for the next iteration.

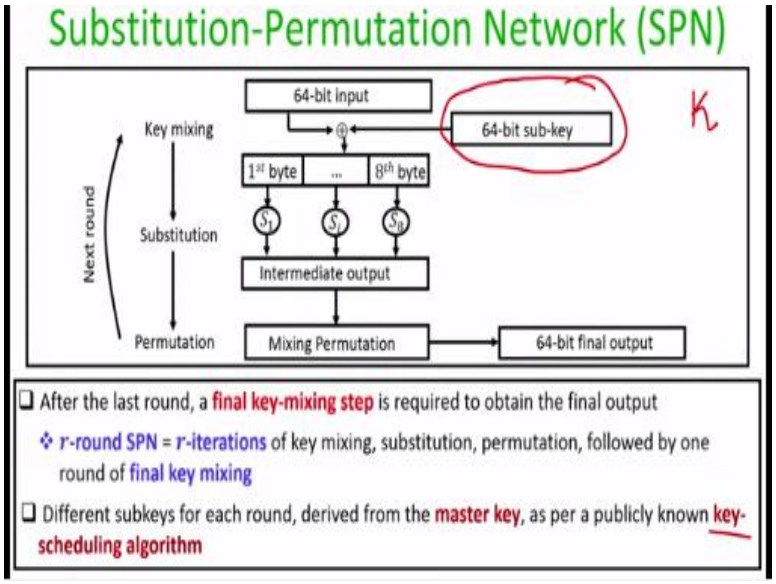
And then we again go through the same process. And after doing it for several iterations, the final outcome is treated as the outcome of the resultant keyed permutation that we have constructed using the SPN architecture ok.

(Refer Slide Time: 19:12)



So if I say that I have a keyed permutation designed using an r-round SPN, then it means that we have done r iterations of key mixing followed by substitution, followed by permutation. And this r-iterations are finally followed by one final round of key mixing, right. So, you might be wondering that why this 1 round of final key mixing after the r^{th} iteration. Well, it turns out that if we do not do this final key mixing at the end of the after the completion of the r^{th} iteration, then the effect of the r^{th} iteration, substitution step and permutation step are completely useless. Because the adversary knows the description of the permutation step and the description of the substitution step. Say if I do not apply this final round or final iteration of the key mixing at the end of the r^{th} iteration. Then it is as good as saying that the effect of r^{th} iteration substitution step and the permutation step is not going to be applicable. So that is why this final key mixing step is useful.

(Refer Slide Time: 20:21)

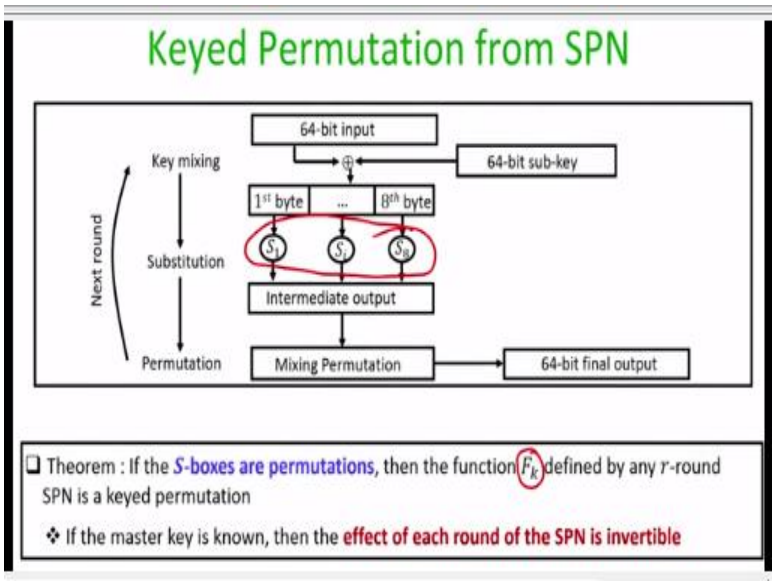


And now, the question comes is how do we determine this sub keys for the individual round. So, we assume that we have a master key k for the function $F_k(x)$ which we are interested to construct. And this 64 bit sub keys are selected from this master key as per a publicly known key scheduling algorithm, right. Say depends upon how exactly my key scheduling algorithm is designed and this key scheduling algorithm will also be publicly known.

Because remember, the only thing which a sender and a receiver who are going to operate this keyed permutation F_k will share is the value of the key. Other than that, we expect that they should not have any pre shared information. So that is why the description of the S boxes, the description of the mixing permutation, the description of the keyed scheduling algorithm everything is available in the public domain.

So, depending upon the length of the master key, we determine a key scheduling algorithm according to which we decide which subset of the master key is going to serve as the sub key for the i^{th} iteration and that description will be publicly known right.

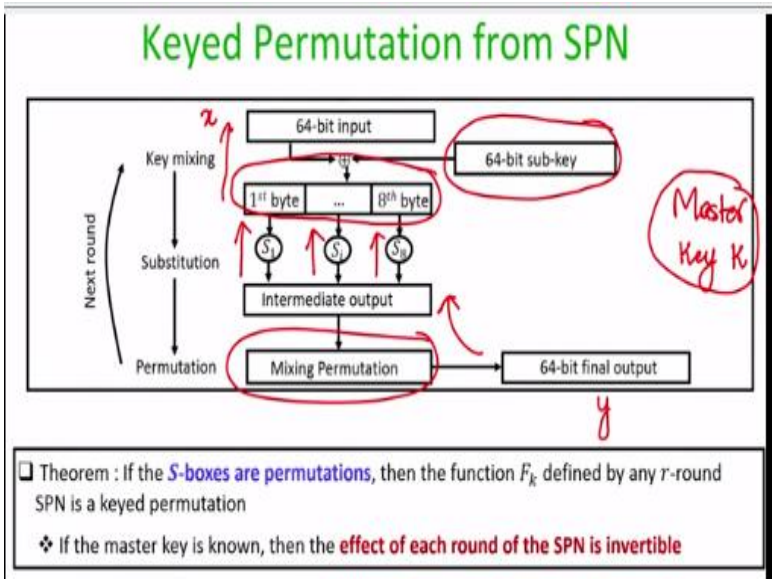
(Refer Slide Time: 21:31)



So, a very important theorem which we can prove is that, if the S boxes which we are using in the architecture of SPN are permutations, then the keyed function F_k which we are going to obtain at the end of any r-round SPN is going to be a permutation. That means it is a one to one and onto mapping and it does not matter what exactly is the value of r. As long as you ensure that these S boxes are invertible, the overall keyed function F_k which we are going to obtain by deploying an r-round SPN is going to be a keyed permutation right.

And the proof for this simply follows from the fact that at the receiving end if the master key is also known to the receiver, then the effect of each round of the SPN is invertible.

(Refer Slide Time: 22:28)



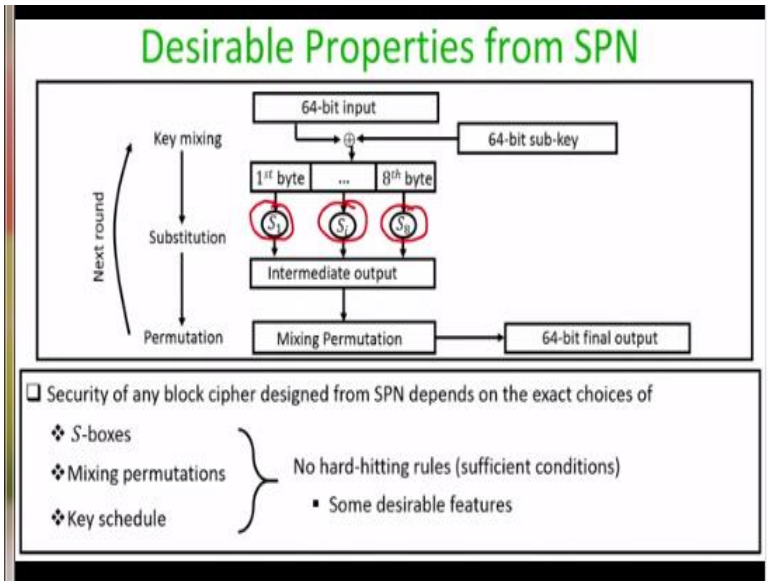
For instance, imagine that the sender has used an input x and it has operated an r-round SPN and it has obtained the value y. And imagine that the receiver knows the value of y and it also knows the master key k right. Now, the goal of the receiver is to uniquely go back from the output y to the input x. So, let us see whether it can reverse back the effect of the last iteration, right.

So, since this mixing permutation is publicly known, from this output y, receiver can go back to this intermediate step. Because the mixing permutation is publicly known, so that as a result in inverse of this output y with respect to this mixing permutation. The input that would have given this output y with respect to this mixing permutation can be uniquely computed because it is a permutation.

Now, once we have reached here, that means we have reached to the state where we know the output of the S boxes 8 S boxes and each of these S boxes are invertible. As a result, we can reverse back the effect of each of these S boxes and we have come to the output of the key mixing stage. And now, if I want to go back from the output of the key mixing stage to the previous input, what we have to do is basically we have to take out or we have to do the XOR of this sub key which I would have used in the last iteration.

And that we can compute provided we know the value of the master key, and the key scheduling algorithm is anyhow publicly available. That means any entity which knows the value of key and y can uniquely invert back the output y and obtain back the input x. And as a result we can say that the overall construction $F_k(x)$ which we have constructed is indeed a keyed permutation.

(Refer Slide Time: 24:24)



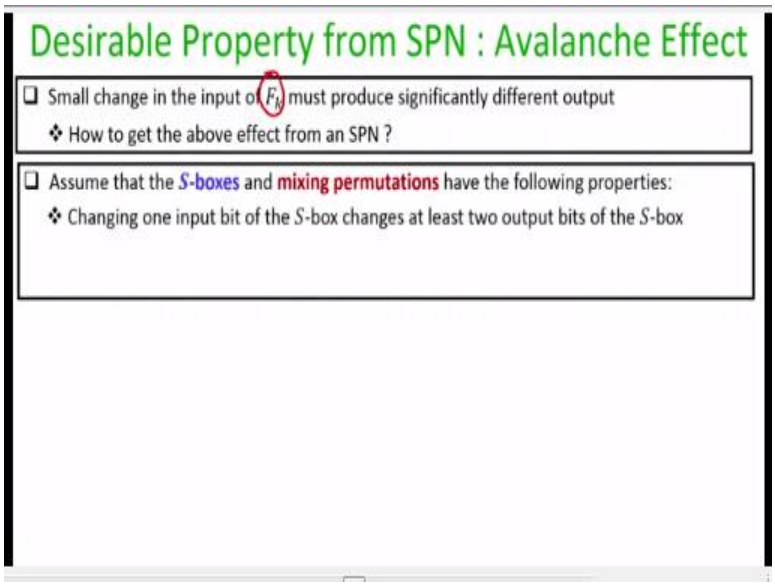
So, now let us come to the security properties of the keyed permutation which we obtained by operating an r-round SPN, right. So, the construction is very straightforward right, now the important thing that we have to discuss is about the security aspect. Now, it turns out that the security of any keyed permutation F_k, which we obtained by running an r-round SPN, depends upon the exact choice of the S boxes which we are using, the exact choice of the mixing permutation that we are using, and the exact choice of key scheduling that we are using right. So in the architecture of SPN, till now I have not discussed what exactly should be the properties of the S boxes. We just required that the S boxes should be invertible to ensure that the overall $F_k(x)$ that we obtain is also invertible. But for the security properties, namely to ensure that your keyed permutation $F_k(x)$ is a pseudo random permutation, we need some desirable properties from these S boxes.

In the same way we need some desirable properties from the mixing permutation and from the key scheduling algorithm right. It turns out that there are no hard hitting rules or sufficient conditions. Namely it is not known that if we ensure certain list of properties with respect to the S boxes mixing permutation and key scheduling algorithm, then the resultant construction is always going to be a pseudo random permutation.

There are no such cookbook or algorithmic rules available. But it turns out that we do expect certain desirable properties from the underlying S boxes, mixing permutations and key

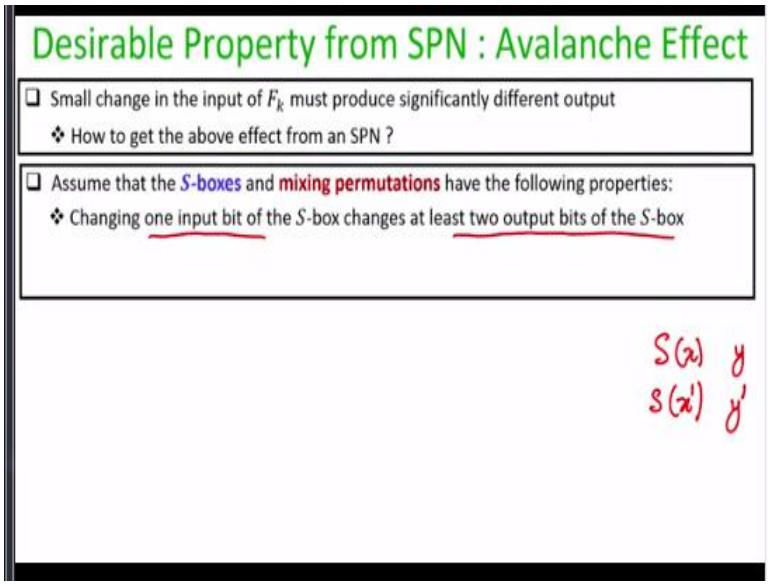
scheduling algorithm to ensure that the resultant keyed permutation indeed behaves like a pseudo random permutations.

(Refer Slide Time: 26: 12)



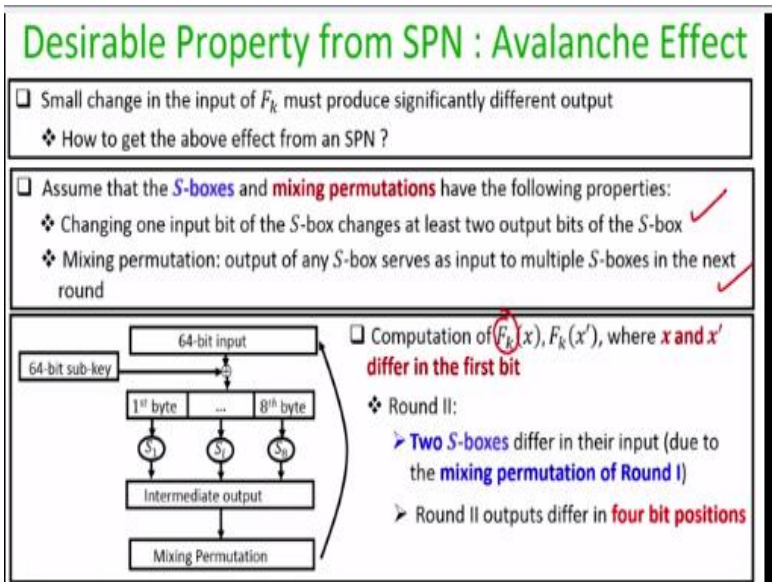
So there is an important property called avalanche effect and what exactly avalanche effect means is that, if we have constructed a keyed permutation F_k using an r-round SPN, then we expect that even a minor change in the input of F_k should produce a significantly different output. Now the question is, how do we ensure that how do choose an SPN which ensures that the resultant F_k designed using the SPN indeed achieves this avalanche effect. So, for the moment assume that we have designed or we are using S boxes the publicly known S boxes and the mixing permutation having the following properties.

(Refer Slide Time: 26:58)



The S boxes that we are using which is publicly known and known to the adversary have the property that : even if there is a change of 1 bit of input in the S box, it ensures that there are at least 2 output bits in the output bits of S box which differ right. That means what I am saying is that assume you have the S box having the property that if you operate it with input x and input x', where x and x' differs only in say 1 bit, then the resultant outputs y and y' differs in at least 2 bits. So for the moment, you assume that all the 8 S boxes that we are using have this property right.

(Refer Slide Time: 27:48)



And for the moment assume that the mixing permutation that we are using have the property that the output of any S box serves as the input to multiple S boxes, when we go to the next iteration.

So remember in the SPN, the mixing permutation basically shuffles the bits of the intermediate output and then only we go to the next iteration. So what we are assuming here is that we have some special type of mixing permutation, which ensures that the output of any S box serves as the input to multiple S boxes in the next round.

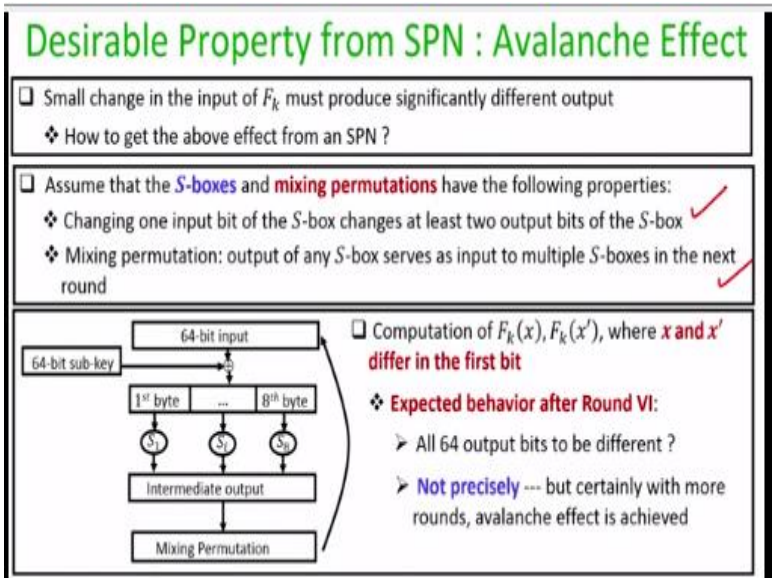
So now let us see how exactly the keyed permutation F_k is going to behave with respect to inputs x and x', where x and x' differs in only say 1 bit say the first bit for simplicity right. Assuming that we are using S boxes in mixing permutations that defines these 2 properties. So it turns out at the end of the first iteration it is only the first S box, whose output is going to differ in 2 bits.

Because x and x' with respect to all the remaining 7 bytes they will be same as a result output of all the 7 S boxes with respect to x and x' will be same. But the value of the output of the first S box with respect to x and with respect to x' will differ in 2 bits. And now, when we apply the output of the first S box with respect to the mixing permutation right, it will be ensured that when we go to the beginning of the second round, there will be 2 S boxes whose inputs will be differing right.

Because this is ensured because of the mixing permutation which we have applied at the end of the first round. That means in round 2 we have now 2 S boxes, whose input differ in at least 1 bit and the output of those 2 S boxes during the second round will differ now in 4 positions. Because we are ensuring that we have our S boxes which ensures that even if there is a change in 1 bit of input the output differs in 2 bit positions.

And now, those 4 bit positions right, where the output of SPN with respect to an x' are differing, they will be going through this special mixing permutation.

(Refer Slide Time: 30:07)

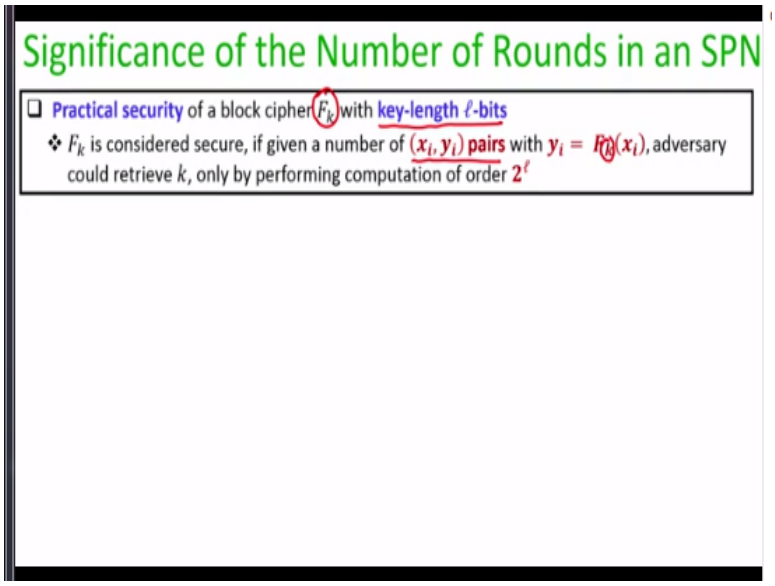


And this will ensure that at the beginning of the third round, we now have several S boxes, whose input bits are differing and so on. So if we continue operating this SPN say for 6 rounds, then ideally we expect that all the 64 intermediate output bits of the SPN are going to be different. But it turns out that ideally we expect this behavior to be achieved by SPN but we would not be able to get it theoretically.

But, it turns out that if we execute the SPN with this special type of S boxes and special type of mixing permutation for sufficiently large number of iterations, then avalanche effect is achieved more or less right. So that gives you an idea that the S boxes and the mixing permutations which we are going to deploy in the SPN. They cannot be any arbitrary S boxes or any arbitrary mixing permutation.

They need to have certain nice properties to ensure that the resultant keyed permutation which we are constructing indeed behave like a truly random permutation or a pseudo random permutation and satisfies the avalanche effect.

(Refer Slide Time: 31:18)



It turns out that the number of rounds inside an SPN is also a very crucial feature and which determines the overall security of the keyed permutation which comes out from the SPN right. So to understand the significance of the number of rounds, it turns out that we have to understand what exactly we mean by the security of the resultant F_k which we are constructing using the SPN, so since we are interested in the practical block cipher, we are not going to measure the security of the resultant block cipher asymptotically rather we define what we mean by a practical security. So imagine the resultant F_k that we are going to obtain by operating an r-round SPN has a key length of l bits right. So then by practical security of F_k, I mean that if adversary got access to several (x_i, y_i) pairs where y_i is the value of F_k on the input x_i and a key is not known to the adversary.

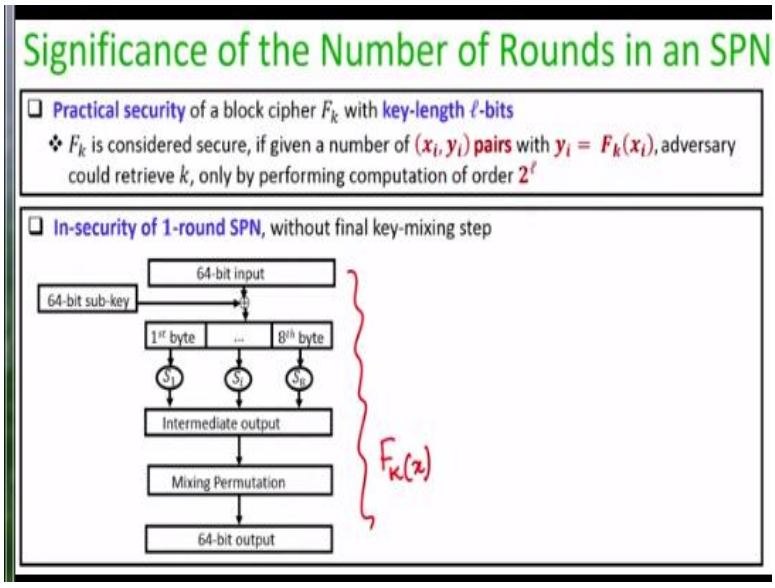
Then the adversary could retrieve the key only by performing computations of order 2^l. If that is the case, then we will say that the resultant block cipher F_k that we have constructed is practically secure. So the idea here is basically we will say the resultant F_k to be secure. If the only possible attack that adversary can launch to recover back the key given several (x_i, y_i) pairs is equivalent to brute force.

If that is the case our F_k will be considered as secure, whereas if there is an attack, whose complexity is of order less than 2^l given several (x_i, y_i) pairs to the adversary, we would not be

considering the resultant F_k to be secure that is the idea here. Now you might be wondering that how exactly the adversary is going to obtain this (x_i,y_i) pairs right.

So remember the CPA game or any CPA secure construction and which uses say a keyed permutation. And in any CPA secure encryption scheme when we play the CPA game with respect to an adversary, adversary is given the access to encryption oracle service. And we have seen in the candidate constructions of CPA secure scheme based on modes of operation that whenever adversary is querying for the encryption oracle service, in response, whatever it our sees it learns the value of the underlying pseudo random permutation on several x values of its choice. That is the way adversary can get access to several x_i, y_i pairs right. So when we are measuring the practical security of block ciphers, we assume that adversary has got access to several x_i, y_i pairs. And its goal is basically to recover back the underlying key; adversary knows the description of the SPN, the only thing that is not known to the adversary is the value of the key and our goal will be to see whether the complexity of the adversary to recover the key is of order that is same as brute force or not right.

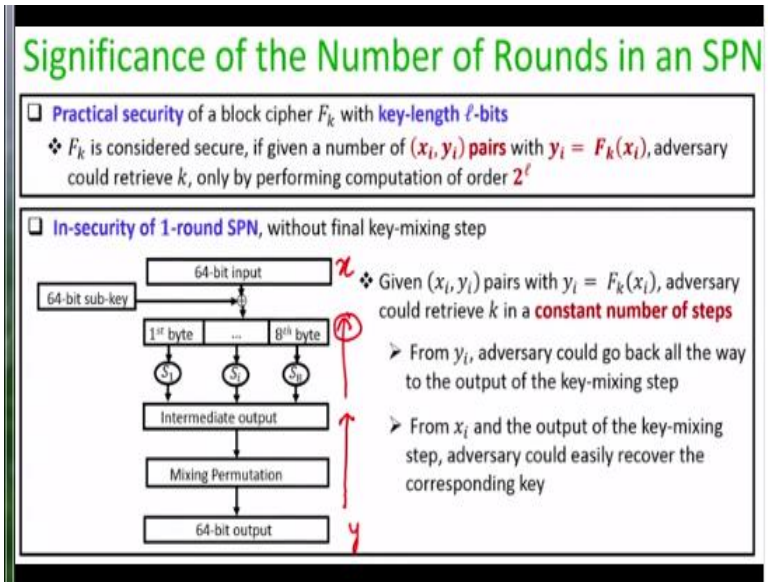
(Refer Slide Time: 34:32)



So let us do a warm up first and see how easy it is for an attacker to simply recover back the key without even doing the brute force, if you just use the 1 round SPN that means imagine someone design a function $F_k(x)$, where function $F_k(x)$ is nothing but 1 round SPN without doing the final key mixing step. So remember, as per the definition of r round SPN, ideally after doing 1

iteration of key mixing, substitution and permutation, there should be the final round of key mixing but that is not happening here.

(Refer Slide Time: 35:09)

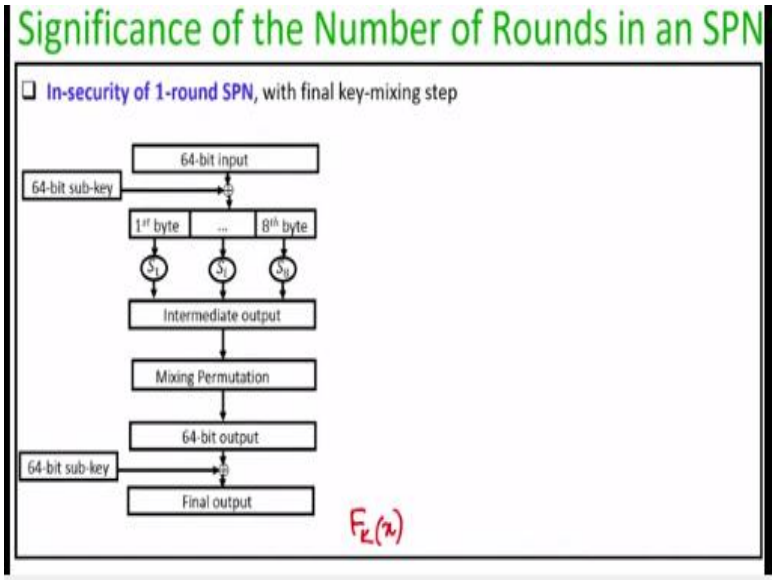


And let us see how trivial it is for an adversary to recover back the key if it has got access to several (x_i, y_i) pairs right. So imagine the adversary knows y, from this y it can go back and reverse back the effect of the mixing permutation go back all the way to the intermediate output here. Namely, the output of the substitution step, it knows the value of the S boxes here right. So it knows the description of the S boxes.

So it can invert back the intermediate output here and it can go back all the way here to the output of the key mixing step. And it knows the x value as well, so since it knows the x value and it knows the output of the key mixing step by XORing the x with the output of the key mixing step, it is easy for the adversary to recover back the sub key. So adversary just have to do constant number of steps here to recover back the complete key.

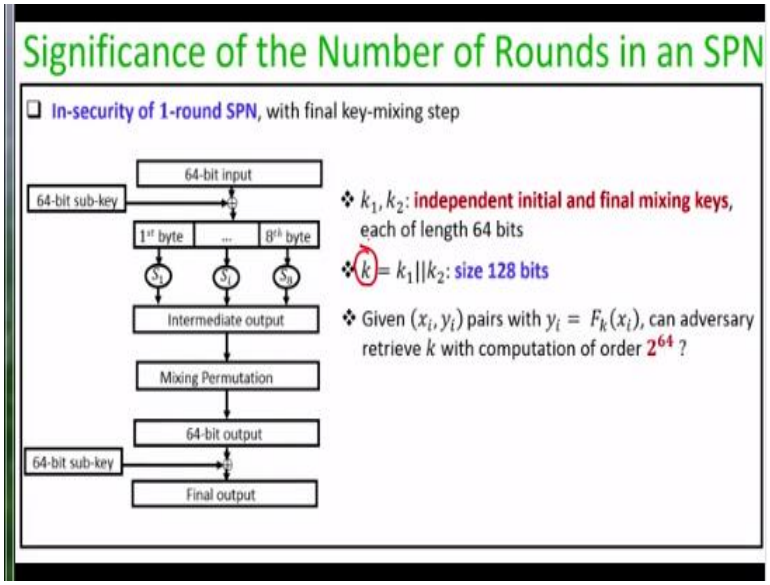
And hence the resultant keyed permutation is not at all secure here, it is very trivial for the adversary to recover back the key.

(Refer Slide Time: 36:15)



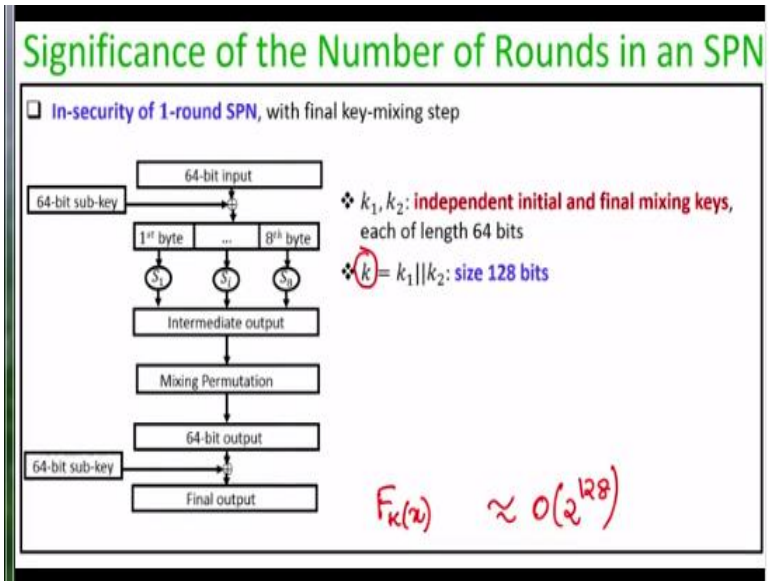
So now let us imagine that we do actual 1 round SPN, namely we design a keyed permutation $F_k(x)$, where we do 1 iteration of key mixing ,substitution and permutation followed by the final key mixing right.

(Refer Slide Time: 36:32)



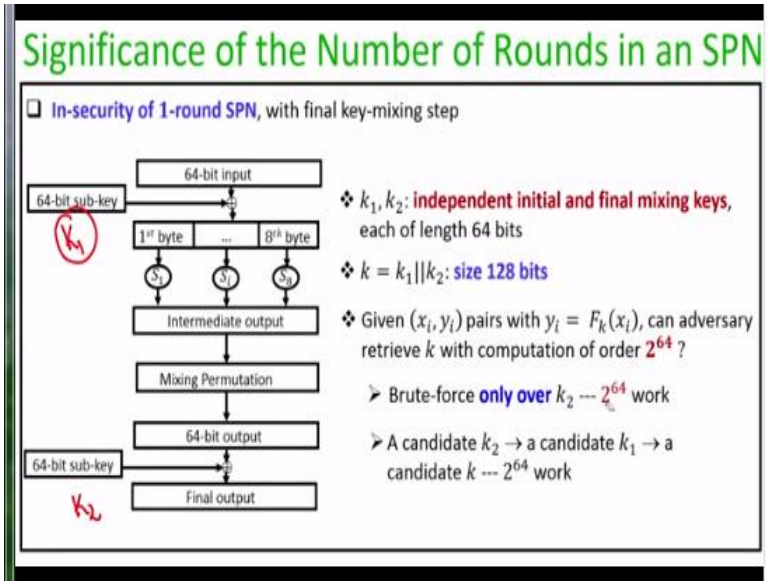
And imagine that the sub keys which are used to during the first round of key mixing and the final key mixing they are independent of each other. And hence the overall master key of my F_k function is of size 128 bits.

(Refer Slide Time: 36:51)



So we will consider the resultant $F_k(x)$ that we have constructed like this to be secure, if the complexity of recovering the unknown key is of order 2^{128} right, because by just doing 2^{128} computation adversary can simply recover back the unknown key if it has got (x,y) pair, where y is the outcome of x or with respect to that unknown key k.

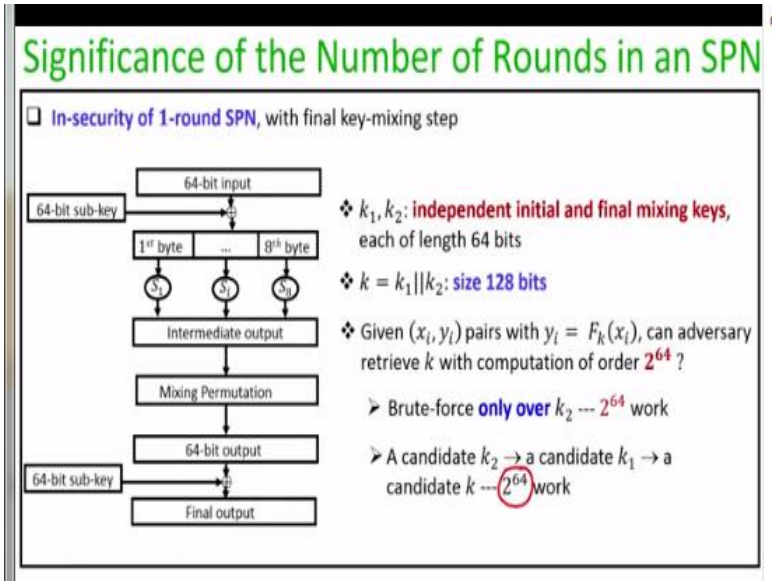
(Refer Slide Time: 37:21)



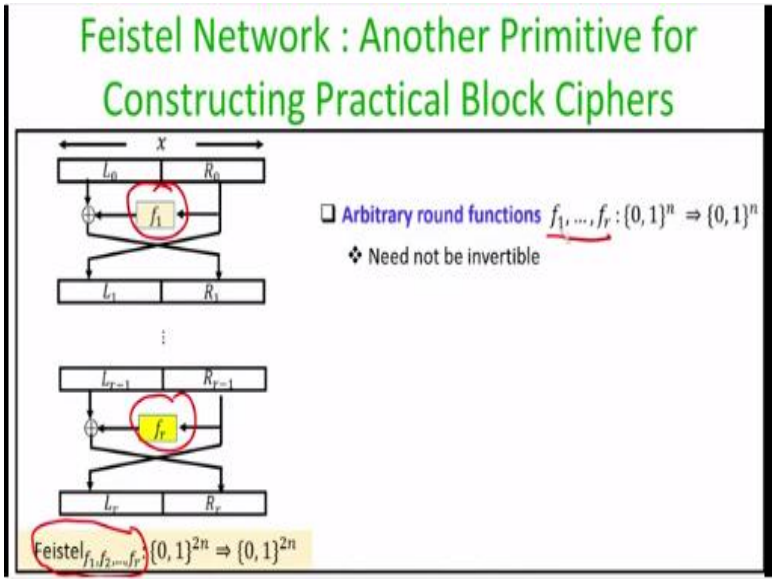
But now what we are going to see is a very interesting attack or a simple attack where adversary can end up actually recovering the 128 bit candidate key by just doing computations of order 2^{64}. And the idea here is that adversary need not have to perform brute force both on the k_2 part as well as on k_1 part. It is sufficient for the adversary to just do a brute force over all candidate k_2's because for each candidate k_2 that adversary thinks in its mind,it gives him 1 candidate k_1 and

that candidate k_1 with respect to the candidate k_2 with the adversary has done the brute force gives him 1 candidate k, that is the overall idea here. So adversary do not need to do an explicit brute force over each candidate k_1 and independently a brute force with each candidate k_2, it suffices for the adversary to just do a brute force over each candidate k_2. So now you can see that if I actually design a keyed permutation using 1 round of SPN and using a key of size 128 bits just by doing computations of order 2^{64}, the adversary can end up recovering a key of size 128 bits and that simply goes against our definition of practical security of block cipher, that means 1 round is not sufficient right.

(Refer Slide Time: 38:25)



(Refer Slide Time: 38:37)



So in practice when we are going to see in the our subsequent lecture the description of the DES it turns out that we have to use an r-round SPN where r is significantly large. In fact in the context of DES the number of iterations that we are going to use is 60. So the number of rounds along with the details of the S boxes, mixing choice of the S boxes, the choice of the mixing permutation, and the choice of the key scheduling algorithm.

All these constituents are very important here, very important aspects of this defining the overall security of the keyed permutation that we design using an r-round SPN. So till now we have discussed SPN which constitutes a very important building block in the design of practical block ciphers. It turns out that there is another important building block namely Feistel network which we also use in the practical constructions of block ciphers.

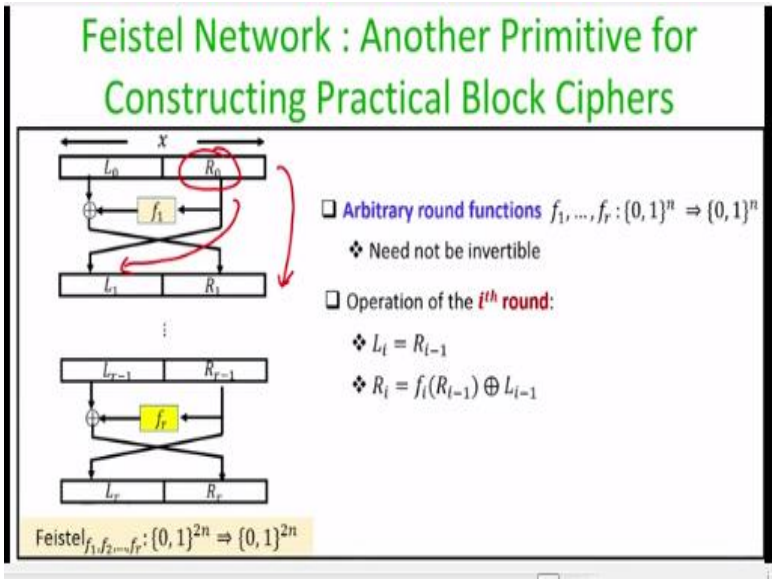
And if you recall, we had already seen the definition of Feistel network, when we have seen how to construct pseudo random permutations from pseudo random function right. So on a very high level what we do in Feistel network is we compose several round functions or a small sized functions in a special way. So if I have composed r-round functions then the overall constructions is denoted by this notation namely Feistel composed with respect to f_1, f_2, f_r.

And the input size of the overall permutation will be 2n bits and output will be 2n bits. The important aspect of the Feistel network or the interesting property of the Feistel network is unlike

SPN where we need the S boxes to be invertible. To ensure that the overall keyed permutation is invertible, when it comes to Feistel networks the round functions that we are using in the individual iterations, they need not be invertible.

It could be any function even if they are not invertible right there is a special way by which we can uniquely invert back the effect of the Feistel network.

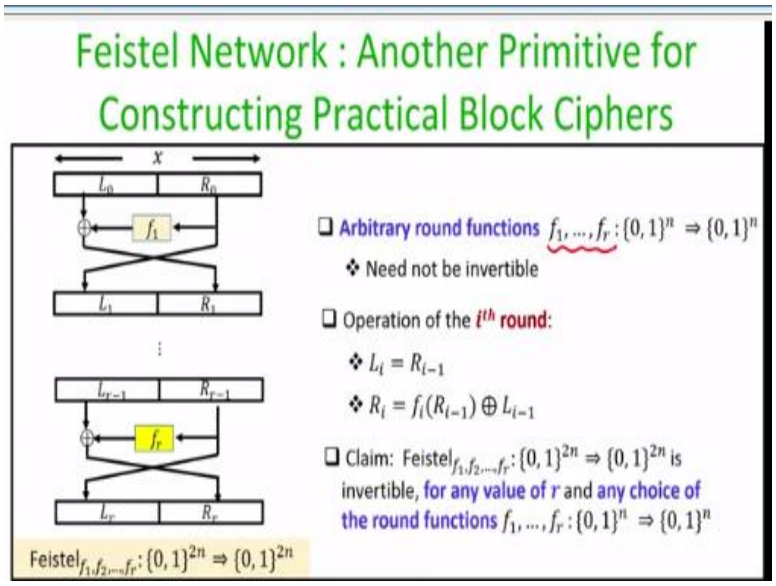
(Refer Slide Time: 40:51)



And to just to recall how exactly the Feistel network operates. If I consider the i^{th} round, say for instance, the first round, the way I go from L_0 R_0 to L_1 R_1 is as follows. My current right half namely R_0, it simply goes and serves as the next L half. And I apply the current round function, namely, f_1 on my current right half and XOR it with my current left half to obtain the next right half.

And this is what we do in every iteration, what differs or what could differ is exactly the choice of the individual round functions. But the operation wise the structure of the operations that we are performing in each iteration is exactly the same.

(Refer Slide Time: 41:41)



And again, I am not going to prove this but we had seen in one of our earlier lectures, that irrespective of what exactly are your individual round functions, for any value of R and any choice of the round functions, the overall function, the composed Feistel network, the overall function that we have obtained by the composed Feistel network is indeed an invertible function.

So, that brings me to the end of this lecture. Just to recall in this lecture, we have seen 2 important building blocks namely SPN and Feistel network, which we use in the constructions of practical block ciphers. And in our next lecture, we will discuss that how these 2 building blocks are used to design some of the real world block ciphers like DES and AES, thank you.